

Lab 4: Simple MicroBlaze Hardware Design

Targeting MicroBlaze™ on Spartan™-3E Starter Kit



Lab 4: Simple Hardware Design Lab

Introduction

This lab guides you through the process of using Xilinx Platform Studio (XPS) to create a simple processor system targeting the Spartan-3E Starter Kit

Procedure

The following diagram represents the completed design (**Figure 1-1**).

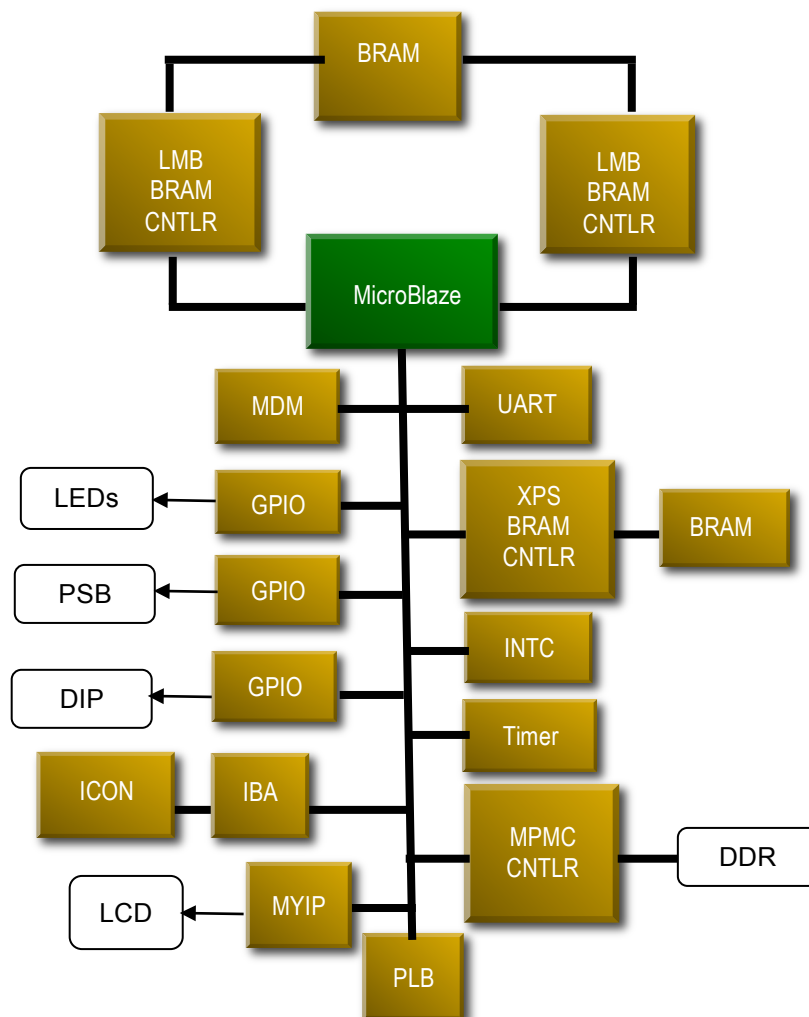


Figure 1-1. Completed Design

In this lab, you will use the BSB of the XPS system to create a processor system consisting of the following processor IP (**Figure 1-2**):

- MicroBlaze (version 7.1)
- PLB_MDM
- LMB BRAM controllers for BRAM
- BRAM
- UART for serial communication
- GPIO for LEDs
- MPMC controller for external DDR_SDRAM memory

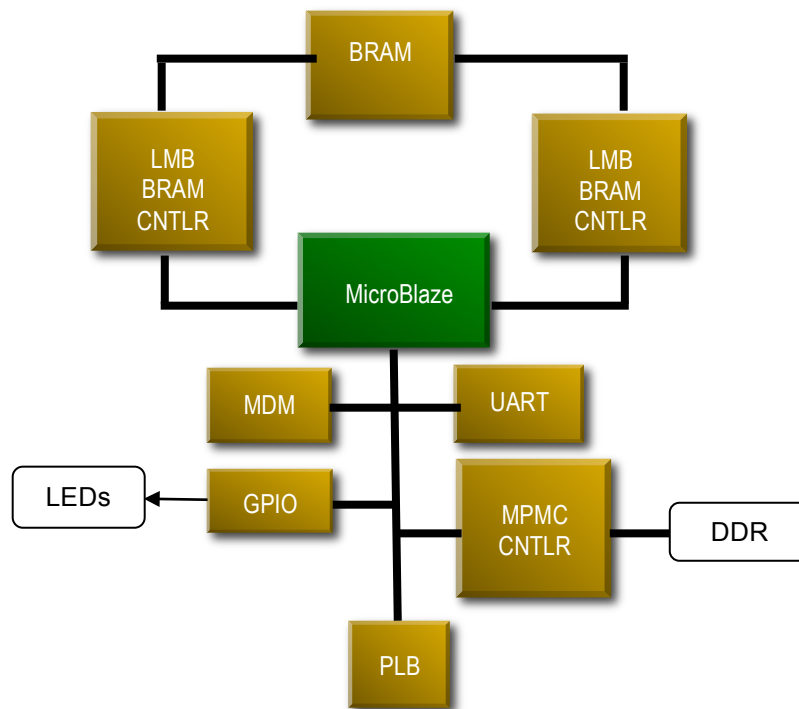


Figure 1-2. Processor IP

This lab comprises three primary steps:

1. Create a project using the Base System Builder
2. Analyze the created project
3. Test in hardware

Creating the Project Using the Base System Builder

Step 1



Launch Xilinx Platform Studio (XPS) and create a new project. Use Base System Builder to generate a MicroBlaze system and memory test application targeting the Spartan-3E starter kit.

- ❶ Open XPS by selecting **Start → Programs → Xilinx ISE Design Suite 10.1 → EDK → Xilinx Platform Studio**
- ❷ Leave the default **Base System Builder** option and click **OK** to start the wizard (**Figure 1-3**). If you clicked cancel, you can select **File → New Project** and the same dialog box will appear.

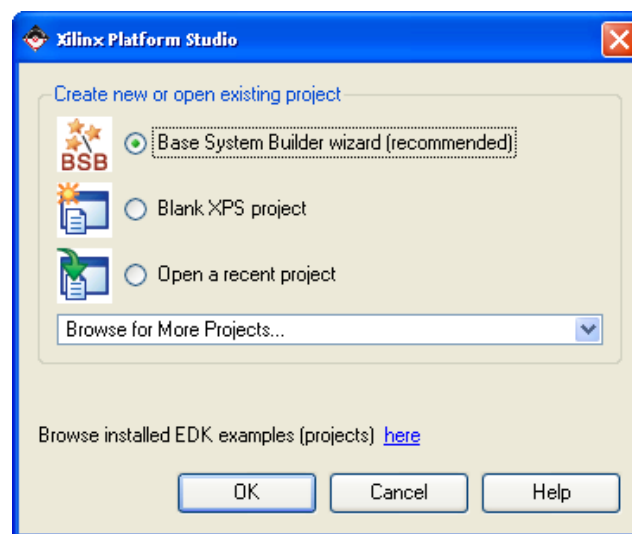


Figure 1-3. New Project Creation Using Base System Builder

- ❸ Browse to **c:\labs** directory, create a new folder called *lab4* and select it, and click **Open** followed by click **Save** (**Figure 1-4**). Click <OK>.

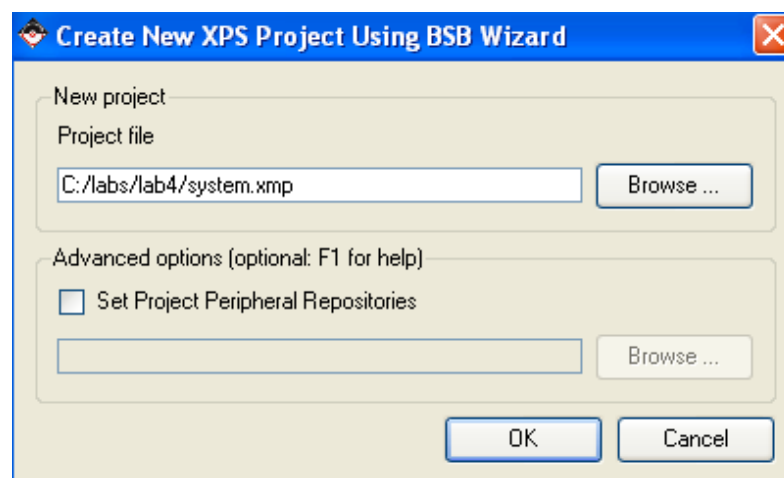


Figure 1-4. Assigning Project Directory

- ❹ Select the **I would like to create a new design option** in the Welcome to Base System Builder dialog box and click **Next**.

- 5 In the **Select Board** dialog box, specify the settings below (**Figure 1-5**) and click **Next** to continue.
 - o Board Vendor: **Xilinx**
 - o Board Name: **Spartan™-3E Starter Board**
 - o Board Revision (Verify on board): **D**

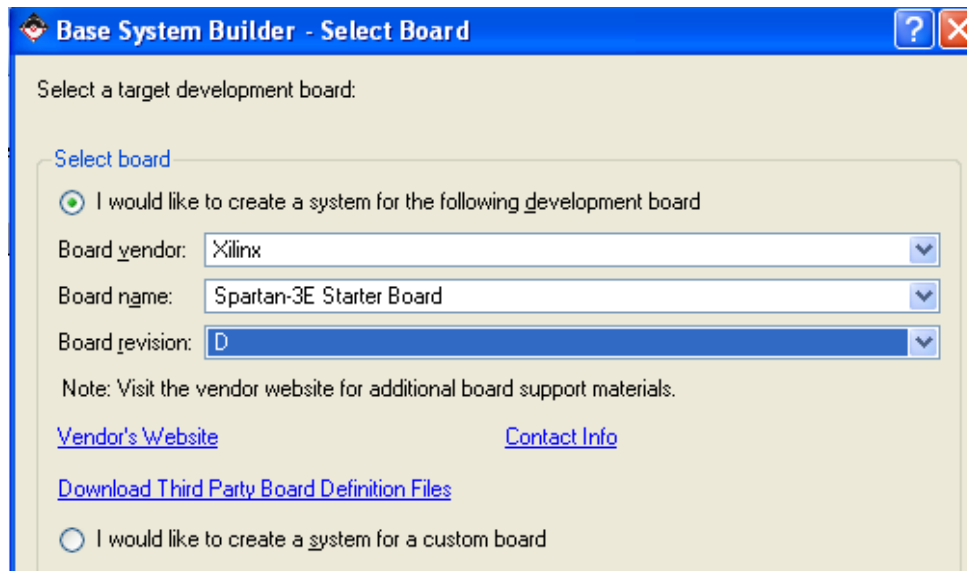


Figure 1-5. Select Board Dialog Box

- 6 In the **Select Processor** dialog, leave the default **MicroBlaze** option (**Figure 1-6**) and click **Next**.

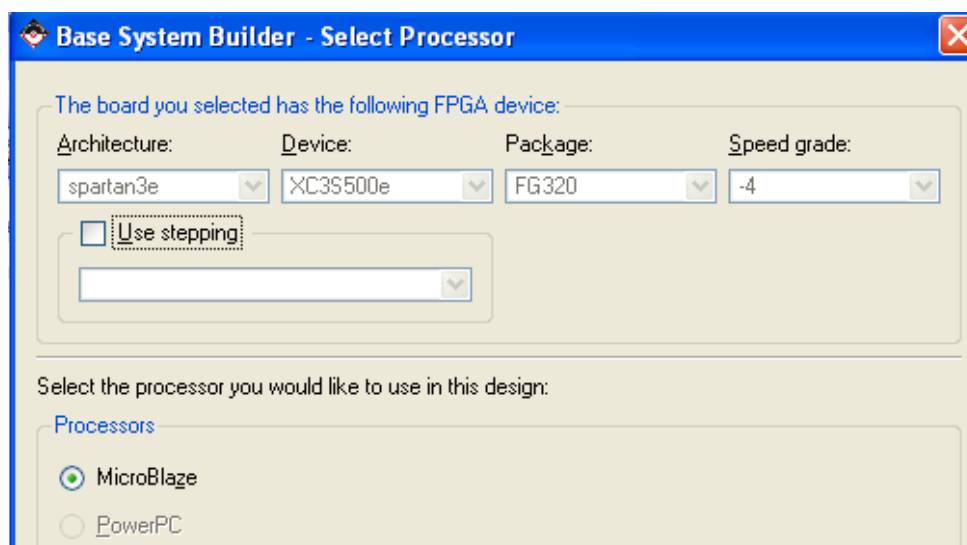


Figure 1-6. Select Processor Dialog Box

- 7 In the **Configure Processor** dialog box (**Figure 1-7**), leave the default settings (see below) and click **Next**.
 - o Reference Clock Frequency: **50 MHz**

- This is the external clock source on the board you are using. This clock will be used to generate the processor and bus clocks.
- Processor –bus Clock Frequency: **50 MHz**
- Debug Interface: **On-Chip H/W debug module**
- Local Data and Instruction Memory – **16 KB**
- Cache Setup: **Enable - unchecked**

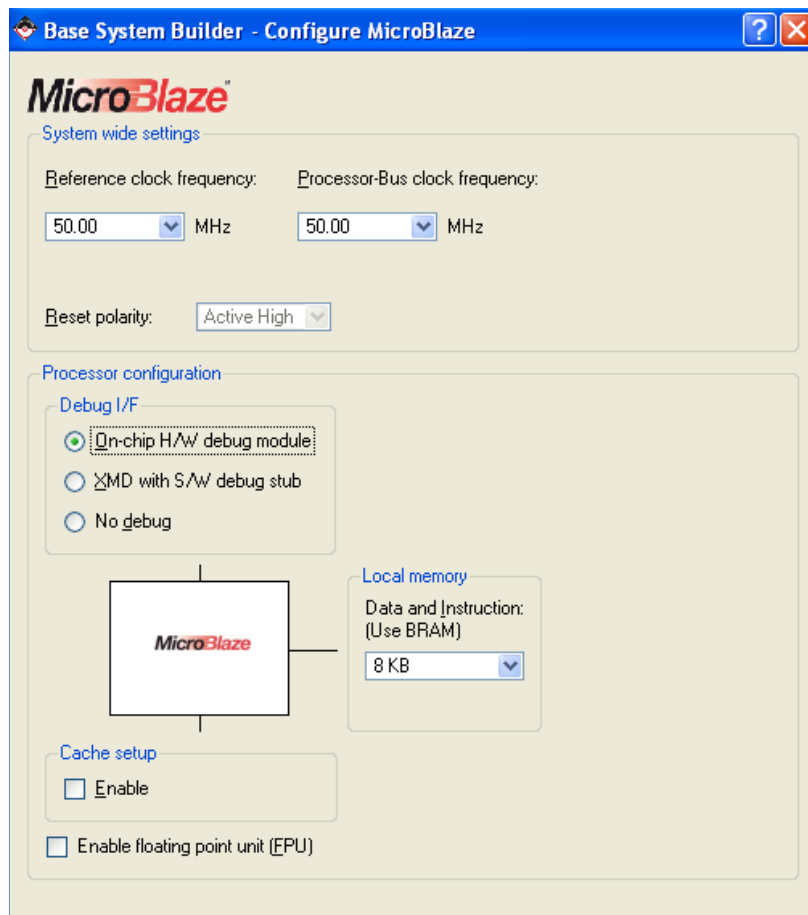


Figure 1-7. Configure Processor Dialog Box



Select and configure the LEDs_8Bit, RS232_DCE, and DDR_SDRAM as the only external devices. Generate the memory test sample application and linker script.

- ❶ In the **Configure IO Interfaces** dialog, select and configure the **RS232_DCE**, **LEDs_8Bit** and **DDR_SDRAM** peripherals as shown below, leaving the rest of the peripherals unchecked.
 - RS232_DCE: XPS UARLITE, 115200 baud rate, 8 Data bits, no interrupt, no parity (**Figure 1-8**)
 - LEDs_8Bit: XPS GPIO. No interrupt (**Figure 1-9**)
 - DDR_SDRAM: MPMC Controller (Multi-Port Memory Controller)

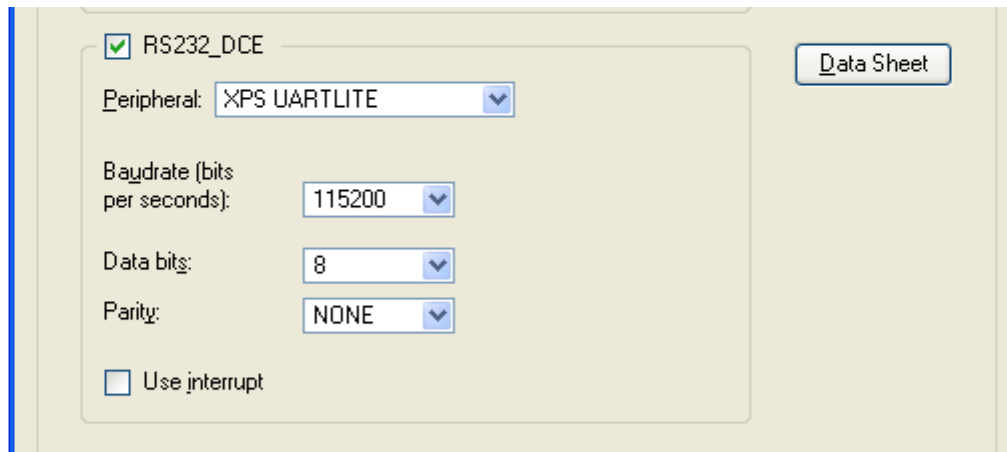


Figure 1-8. Configure RS-232 DCE

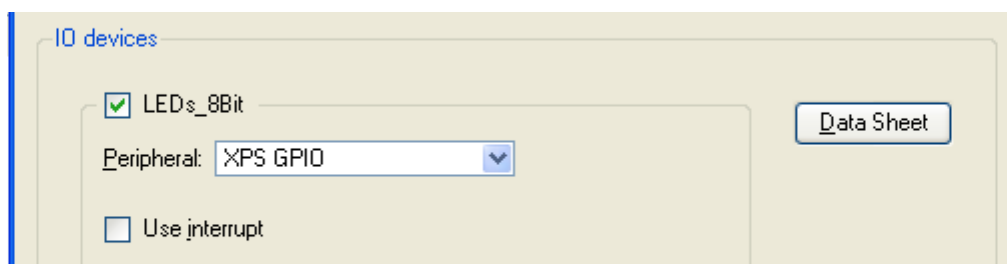


Figure 1-9. Configure XPS GPIO

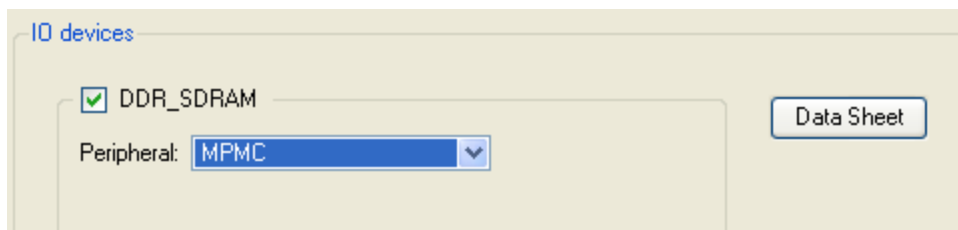


Figure 1-10: Configure DDR_SDRAM with MPMC Controller

- 2 Click **Next** until the **Add Internal Peripherals** dialog is displayed, making sure that none of the other devices are selected

At this point you could click **Add Peripheral** to add additional internal peripherals, but you will see an alternative method in the next lab for adding internal peripherals to an existing project.

- 3 Click **Next** to display the **Software Setup** dialog box.
- 4 Unselect Peripheral selftest (**Figure 1-11**) and click **Next**.

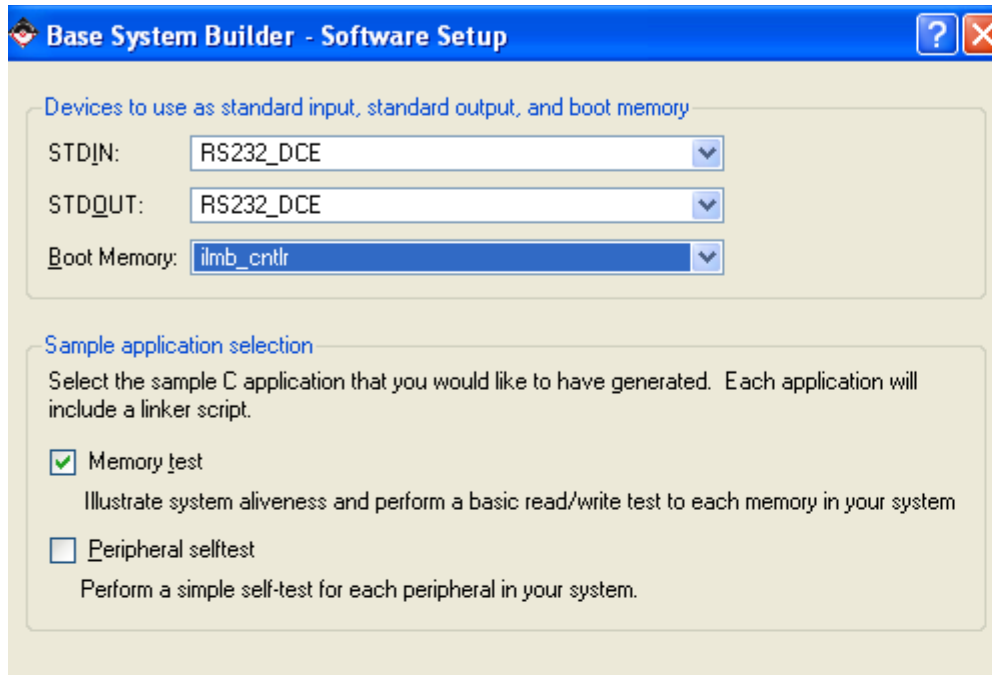


Figure 1-11. Software Setup Dialog Box

- ⑤ Leave the default selections in the **Configure Memory Test Application** dialog (Figure 1-12) and click **Next**.

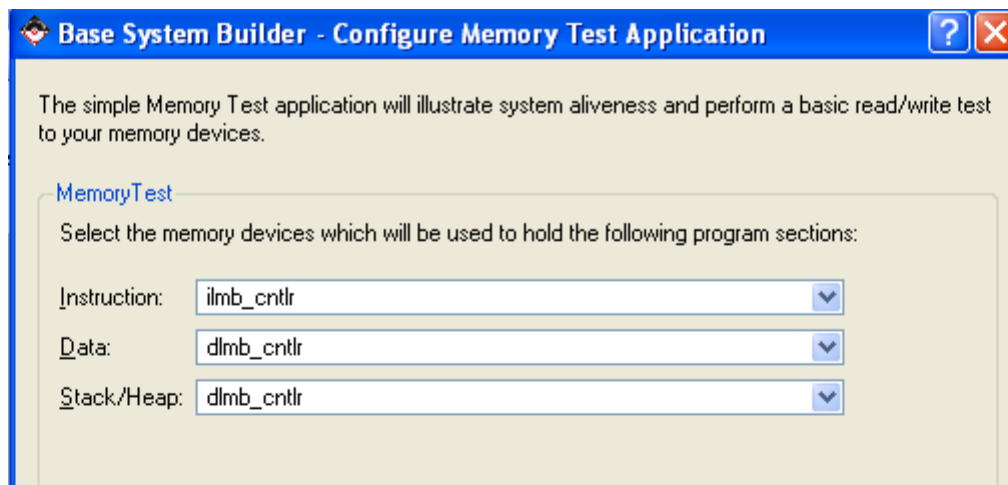


Figure 1-12. Configure Memory Test Application

- ⑥ Verify the system summary in the **System Created** dialog (**Figure 1-13**) and click **Generate**

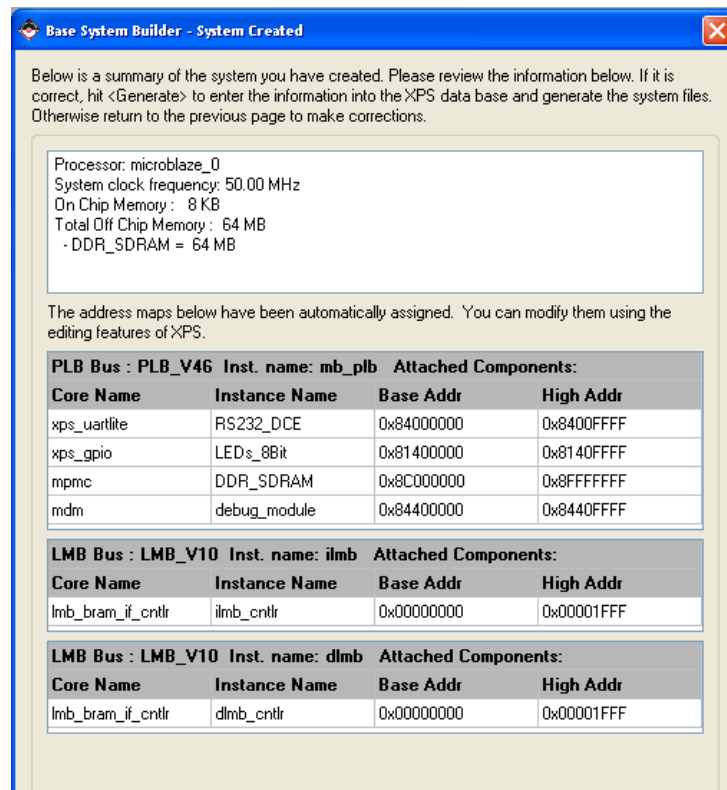


Figure 1-13. System Created Dialog Box

- ⑦ Click **Finish** once the congratulations dialog box appears, indicating the files that BSB has created.

- ⑧ In the **Next Step** dialog box, ensure **Start Using Platform Studio** is checked and click **OK**

A Software Agreement dialog may appear if this is the first time the software is run

- ⑨ A System Assembly View1 will be displayed (**Figure 1-14**) showing peripherals and busses in the system, and the system connectivity.

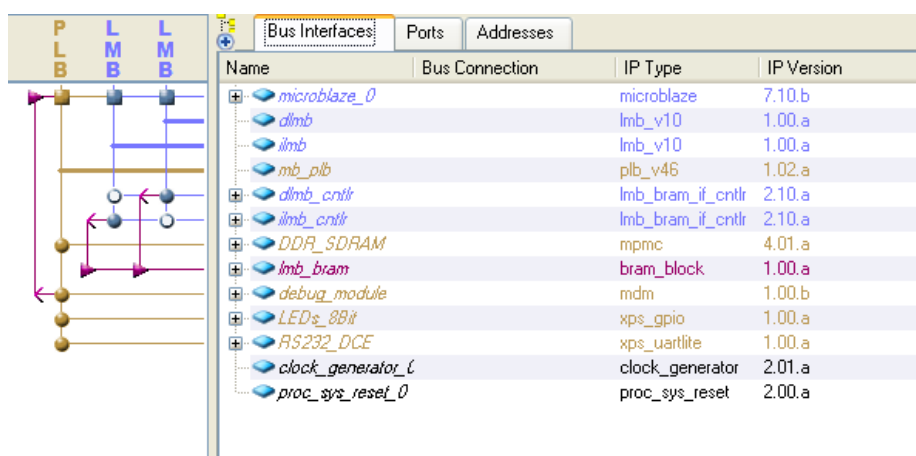


Figure 1-14. System Created Dialog Box

Analyze the Hardware

Step 2



Generate a block diagram of the system and study the system components and interconnections. Look in the System Assembly View and analyze the bus and port connections. Run PlatGen to generate the system netlists (NGC) and review the generated files.

- Click on the **Block Diagram tab** to open a block diagram view (**Figure 1-15**) and observe the various components that are used in the design

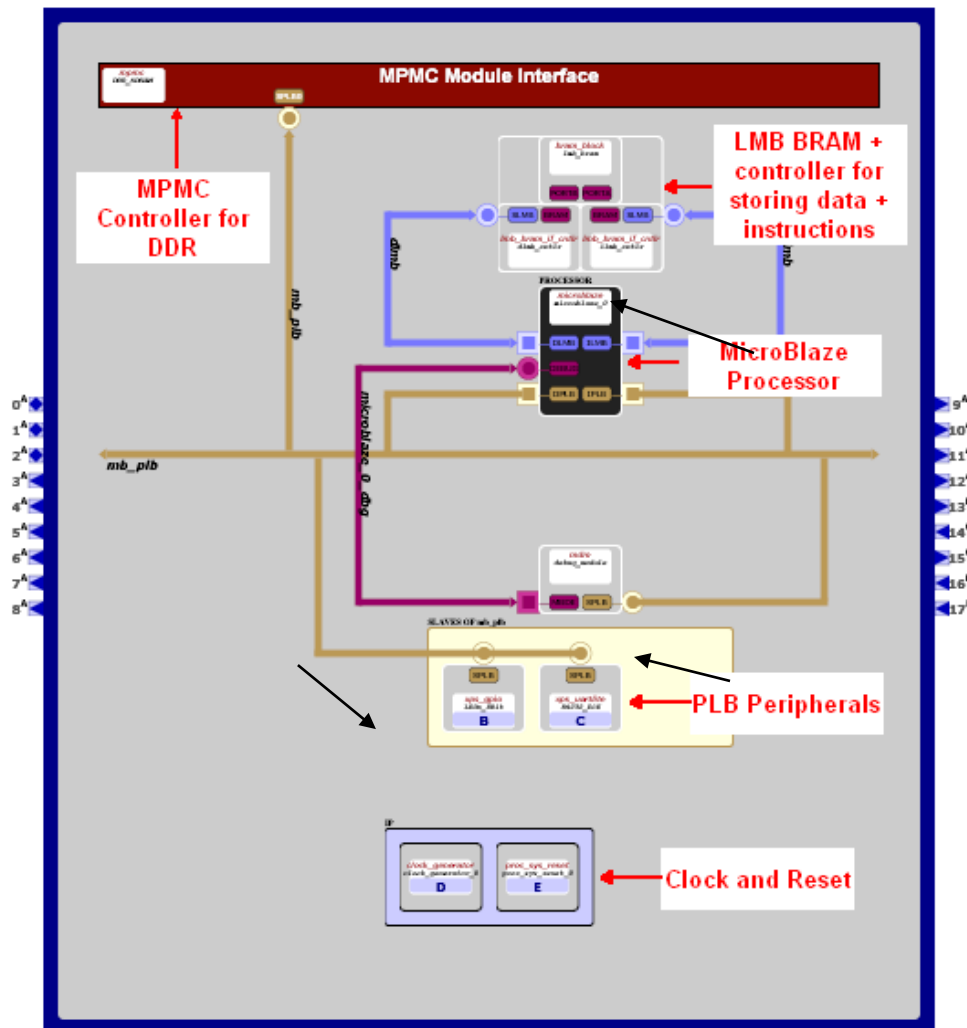


Figure 1-15. Block Diagram View of the Generated Project

You can zoom in and out and use the scroll bars to navigate around the block diagram. You will see the MicroBlaze™ processor, LMB controller and PLB bus connected to the MicroBlaze processor. In addition, you will see the I/O ports on the sides and legend at the bottom of the diagram.

- In the **System Assembly View** click on plus button and observe the expanded (detailed) bus connection view of the system (**Figure 1-16**)

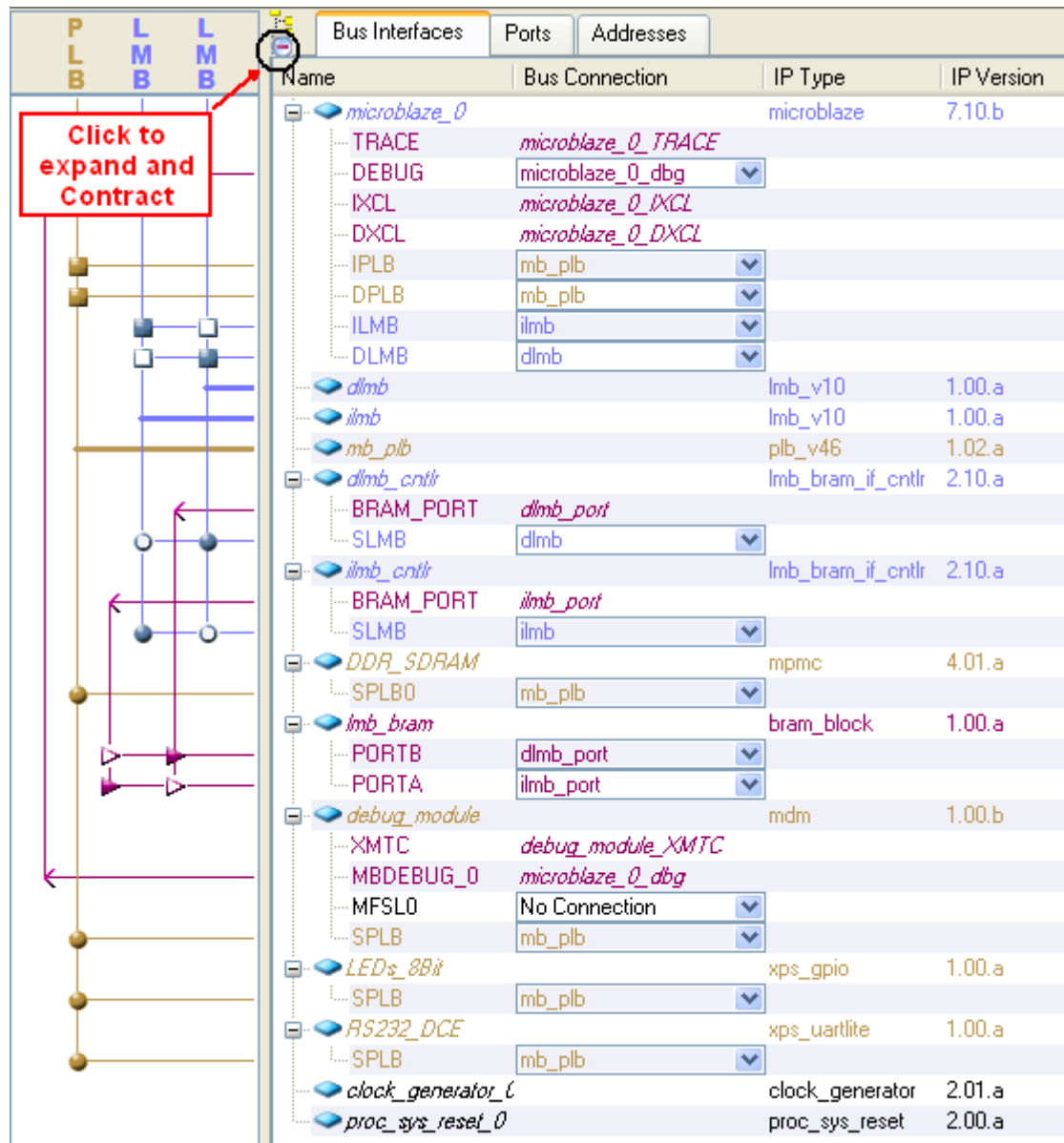


Figure 1-16. Detailed Bus Connections

- Click on the **Ports** filter and have an expanded view similar to **Figure 1-17**. This is where you can make internal and external net connections.


Bus Interfaces Ports Addresses			
Name	Net	Direction	Range
External Ports			
sys_rst_pin	sys_rst_s	I	
sys_clk_pin	dcm_clk_s	I	
fpga_0_DDR_SDRAM_DDR...	fpga_0_DDR_SDRAM_DDR_DQ	IO	[15:0]
fpga_0_DDR_SDRAM_DDR...	fpga_0_DDR_SDRAM_DDR_DQS	IO	[1:0]
fpga_0_DDR_SDRAM_DDR...	fpga_0_DDR_SDRAM_DDR_DM	O	[1:0]
fpga_0_DDR_SDRAM_DDR...	fpga_0_DDR_SDRAM_DDR_WE_n	O	
fpga_0_DDR_SDRAM_DDR...	fpga_0_DDR_SDRAM_DDR_RAS_n	O	
fpga_0_DDR_SDRAM_DDR...	fpga_0_DDR_SDRAM_DDR_CS_n	O	
fpga_0_DDR_SDRAM_DDR...	fpga_0_DDR_SDRAM_DDR_CE	O	
fpga_0_DDR_SDRAM_DDR...	fpga_0_DDR_SDRAM_DDR_CAS_n	O	
fpga_0_DDR_SDRAM_DDR...	fpga_0_DDR_SDRAM_DDR_BankAc	O	[1:0]
fpga_0_DDR_SDRAM_DDR...	fpga_0_DDR_SDRAM_DDR_Addr	O	[12:0]
fpga_0_DDR_SDRAM_DDR...	fpga_0_DDR_SDRAM_DDR_Clk_n	O	
fpga_0_DDR_SDRAM_DDR...	fpga_0_DDR_SDRAM_DDR_Clk	O	
fpga_0_DDR_SDRAM_DDR...	fpga_0_DDR_SDRAM_DDR_DQS_D	IO	
fpga_0_LEDs_8Bit_GPIO_d_o...	fpga_0_LEDs_8Bit_GPIO_d_out	O	[0:7]
fpga_0_RS232_DCE_TX_pin	fpga_0_RS232_DCE_TX	O	
fpga_0_RS232_DCE_RX_pin	fpga_0_RS232_DCE_RX	I	
microblaze_0			
MB_Halted	No Connection	O	
DBG_STOP	No Connection	I	
INTERRUPT	No Connection	I	
MB_RESET	mb_reset	I	
dlmb			
SYS_Rst	sys_bus_reset	I	
LMB_Clk	sys_clk_s	I	

Figure 1-17. Ports Filter

- Click on the **Addresses** tab and have an expanded view similar to **Figure 1-18**. This is where you can assign base/high addresses to the peripherals in the system.

Bus Interfaces Ports Addresses							
Bus Interfaces		Ports		Addresses			
Name	Base Address	High Address	Size	Bus Interface(s)	Bus Connection	Lock	
mb_plb							
dlmb_cntlr							
C_BASEADDR	0x00000000	0x00001fff	8K	SLMB	dlmb	<input type="checkbox"/>	
ilmb_cntlr							
C_BASEADDR	0x00000000	0x00001fff	8K	SLMB	ilmb	<input type="checkbox"/>	
DDR_SDRAM							
C_MPMC_BASEADDR	0x8c000000	0x8fffffff	64M	SPLB0	mb_plb	<input type="checkbox"/>	
debug_module							
C_BASEADDR	0x84400000	0x8440ffff	64K	SPLB	mb_plb	<input type="checkbox"/>	
LEDs_8Bit							
C_BASEADDR	0x81400000	0x8140ffff	64K	SPLB	mb_plb	<input type="checkbox"/>	
RS232_DCE							
C_BASEADDR	0x84000000	0x8400ffff	64K	SPLB	mb_plb	<input type="checkbox"/>	

Figure 1-18. Assign Base/High Addresses

- ⑤ Run PlatGen by selecting **Hardware** → **Generate Netlist** or click  in the toolbar

Test in Hardware

Step 3



Generate bitstream and download to the board. Prior to download, the instruction memory (FPGA Block RAM) will be updated in the bitstream with the executable generated using the GNU compiler.

- ① Connect and power up the Spartan-3E starter kit
- ② Open a hyperterminal session (**Figure 1-19**)

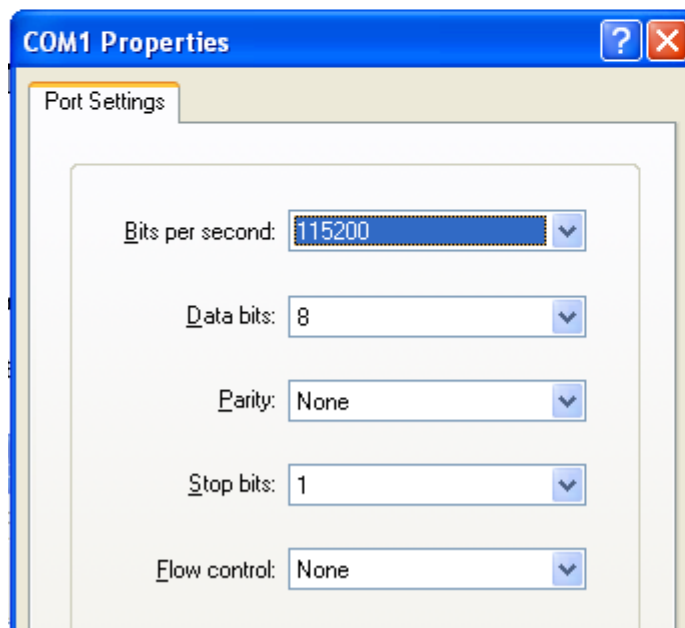


Figure 1-19. HyperTerminal Settings

- ③ Select **Device Configuration** → **Download Bitstream** in XPS.

You should see the following output on hyperterminal

```
-- Entering main() --
Starting MemoryTest for DDR_SDRAM:
  Running 32-bit test...PASSED!
  Running 16-bit test...PASSED!
  Running 8-bit test...PASSED!
-- Exiting main() --
```

Figure 1-20. HyperTerminal Output

Adding IP to a Hardware Design Lab

Step 4

The purpose of this step is to extend the hardware design (Figure 1-21) created according to the following procedure

1. Add and connect GPIO peripherals in the system
2. Configure the GPIO peripherals
3. Make external GPIO connections
4. Analyze the MHS file
5. Add the software application and compile
6. Verify the design in hardware

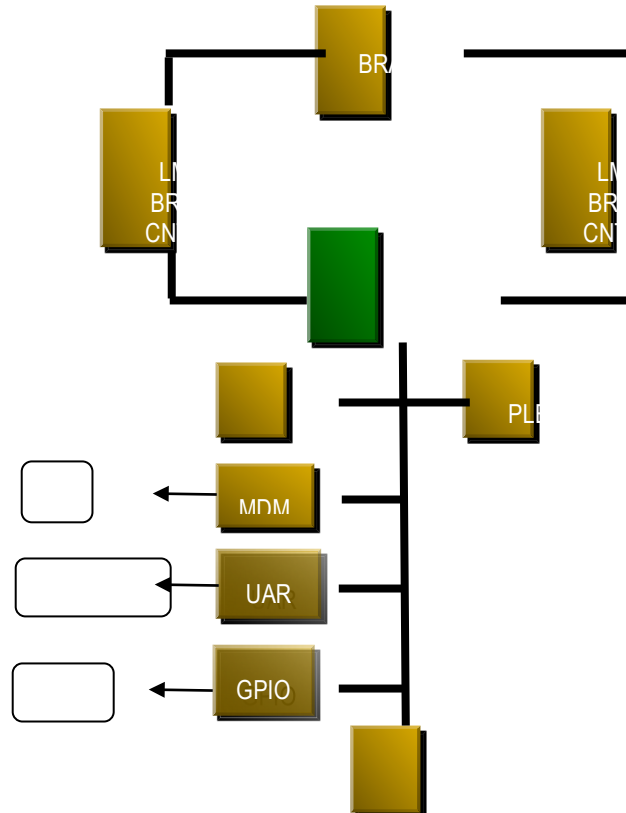


Figure 1-21. Extend the System from the previous step

Add and Connect GPIO Peripherals to the System

Step 5



Add two instances of an XPS GPIO Peripheral from the IP catalog to the processor system via the System Assembly View.

XPS provides two methods for adding peripherals to an existing project. You will use the first method, the System Assembly View panel, to add most of the additional IP and connect them. The second method is to manually edit MHS file.

- ❶ Select the **IP Catalog** tab in the left window and click on plus sign next to **General Purpose IO** entry to view the available cores under it (Figure 1-22)

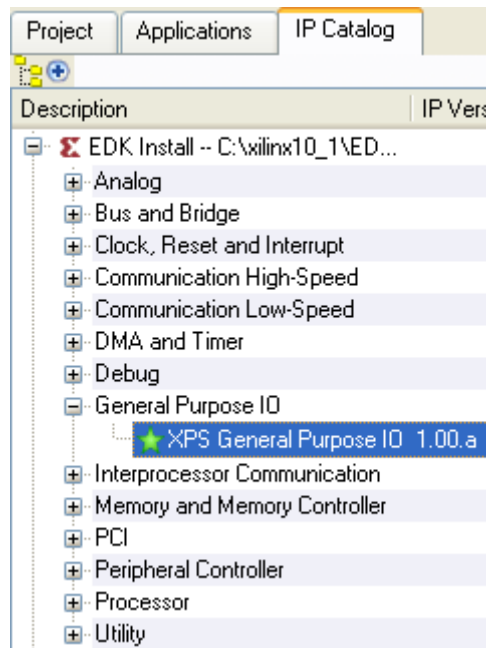


Figure 1-22. System Assembly View

- 2 Double-click on the **XPS General Purpose IO** core twice, to add two instances to the System Assembly View
- 3 Change the instance names of the peripherals to **dip** and **push**, by clicking once in the name column, typing the new name for the peripheral followed by pressing Enter key

At this point, the System Assembly View should look like the following (Figure 1-23):

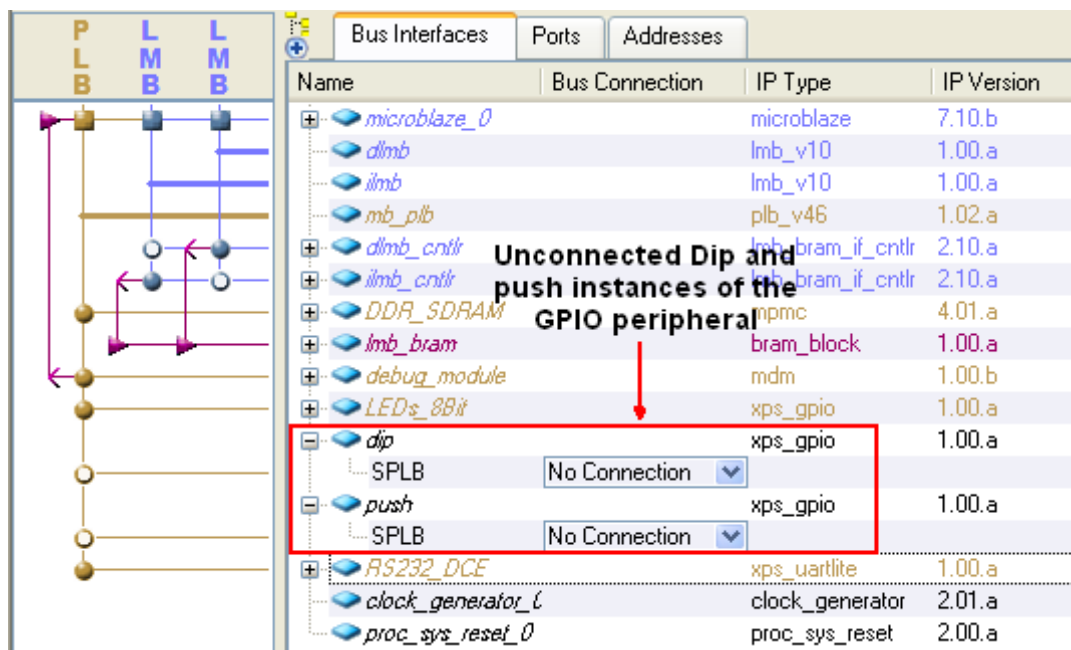


Figure 1-23. System Assembly View After Adding Peripherals

- 4 Click once in **Bus Connection** column for the **push** and **dip** instances to connect them as slave devices to the PLB.

At this point, the Bus Connections tab should look like the following (**Figure 1-24**):

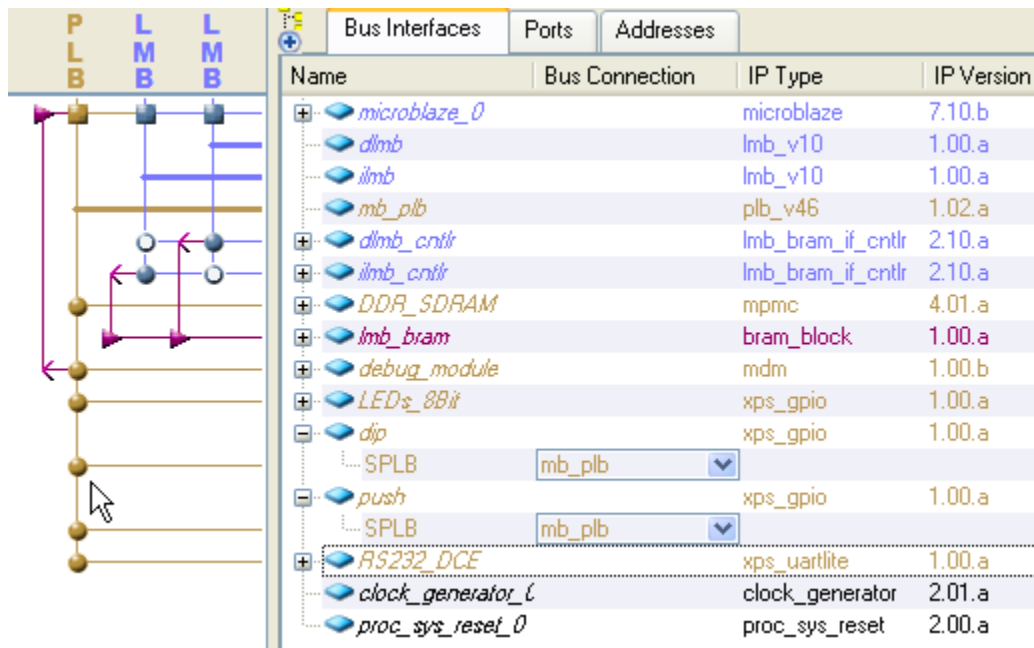


Figure 1-24. Bus Interfaces Tab showing Bus Connections to the Added Peripherals

- 5 Select the **Addresses** filter

You can manually assign the base address and size of your peripherals or have XPS generate the addresses for you.

- 6 Click under the **size** column in the **push** and **dip** instances, change it to **64K**, and hit Enter key
- 7 Click **Generate Addresses** (located on the right most end of the tabs) to automatically generate the base and high addresses for the peripherals in the system. The base address and high addresses will change as shown in **Figure 1-25** below

Bus Interfaces Ports Addresses						
Instance	Name	Base Address	High Address	Size	Bus Interface(s)	
dlmb_cntlr	C_BASEADDR	0x00000000	0x00001fff	8K	SLMB	
ilmb_cntlr	C_BASEADDR	0x00000000	0x00001fff	8K	SLMB	
debug_module	C_BASEADDR	0x84400000	0x8440ffff	64K	SPLB	
dip	C_BASEADDR	0x81420000	0x8142ffff	64K	SPLB	
push	C_BASEADDR	0x81400000	0x8140ffff	64K	SPLB	
LEDs_8Bit	C_BASEADDR	0x81440000	0x8144ffff	64K	SPLB	
RS232_DCE	C_BASEADDR	0x84000000	0x8400ffff	64K	SPLB	
DDR_SDRAM	C_MPMC_BASEADDR	0x8c000000	0x8fffffff	64M	SPLB0	

Figure 1-25. Peripherals Memory Map

Configure the GPIO Peripherals

Step 6



There are four push buttons and four DIP switches on the Spartan-3E starter kit. You will first configure the **push** and **dip** instances according to their sizes and direction, and then make external pin connections.

- ❶ Select the **Ports** filter in the toolbar of the System Assembly View
- ❷ Double-click on the **push** instance to access the configuration window

Notice that the peripheral can be configured for two channels, but, since we want to use only one channel leave the **Enable Channel 2** *unchecked*.

- ❸ Click on the **GPIO Data Bus Width** down arrow and set it to **4**, you will use 4 push buttons on the Spartan-3E starter kit.

The settings for the Common parameters should be set according to **Figure 1-26** below.

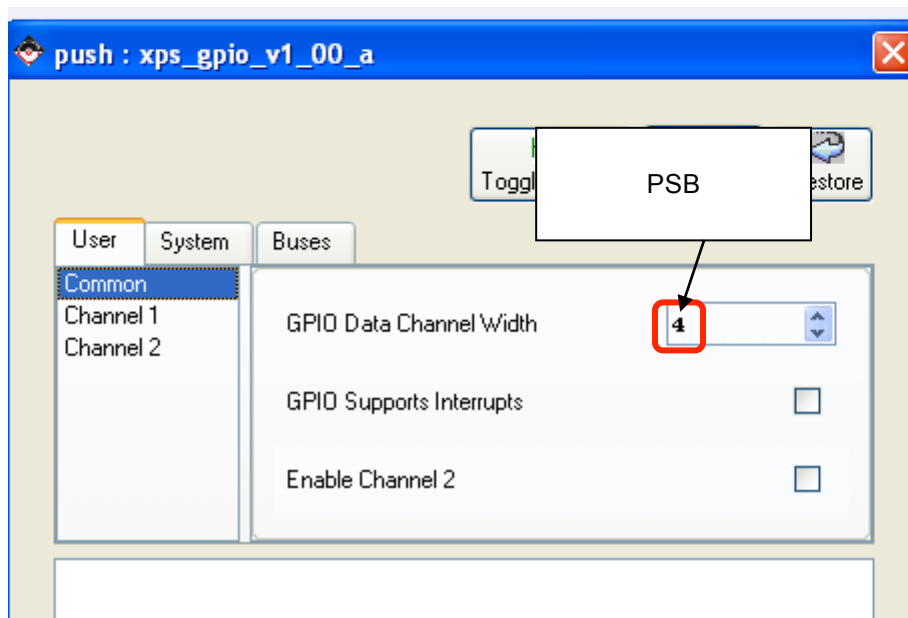


Figure 1-26. Configurable Parameters of GPIO Instance for Push Buttons

- ❹ Next click **Channel 1** and set **Channel 1 is Bi-directional** to *False* and **Channel 1 is input Only** to *True* (**Figure 1-27**):

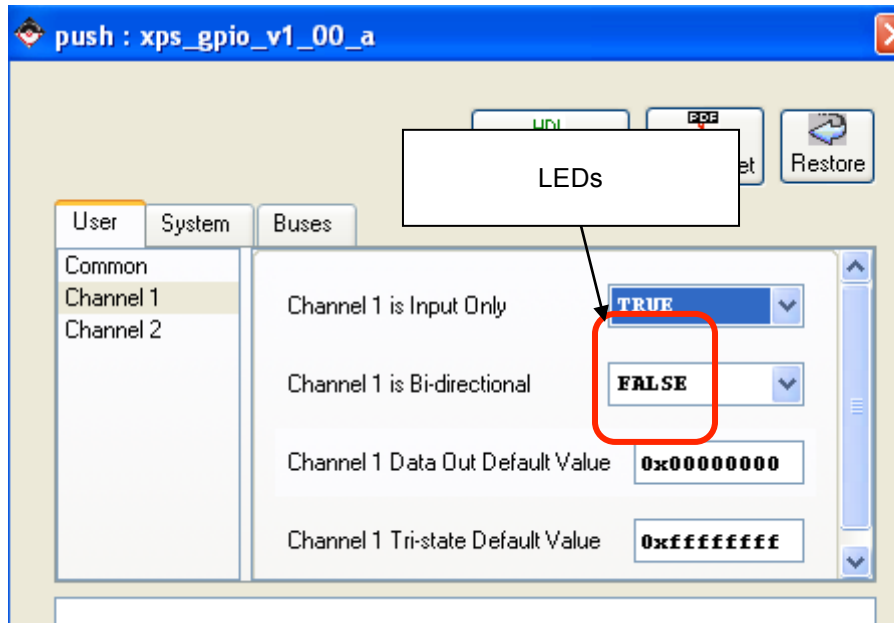


Figure 2-7. Setting Configurable Parameters for Push Buttons

- ⑥ Set the same parameters for the **dip** instance, as performed for the push buttons.

Make External GPIO Peripheral Connections

Step 7



You will connect the **push** and **dip** instances to the push buttons and DIP switches on the Spartan-3E starter kit. In order to do this, you must establish the GPIO data ports as external FPGA pins and then assign them to the proper locations on the FPGA via the UCF file. The location constraints are provided for you in this section. Normally, one would consult the Spartan-3E starter kit user manual to find this information.

- ① Make the **GPIO_in** port of the **push** instance as external by selecting **Make External**. You should see a new external net connection (**Figure 1-28**).

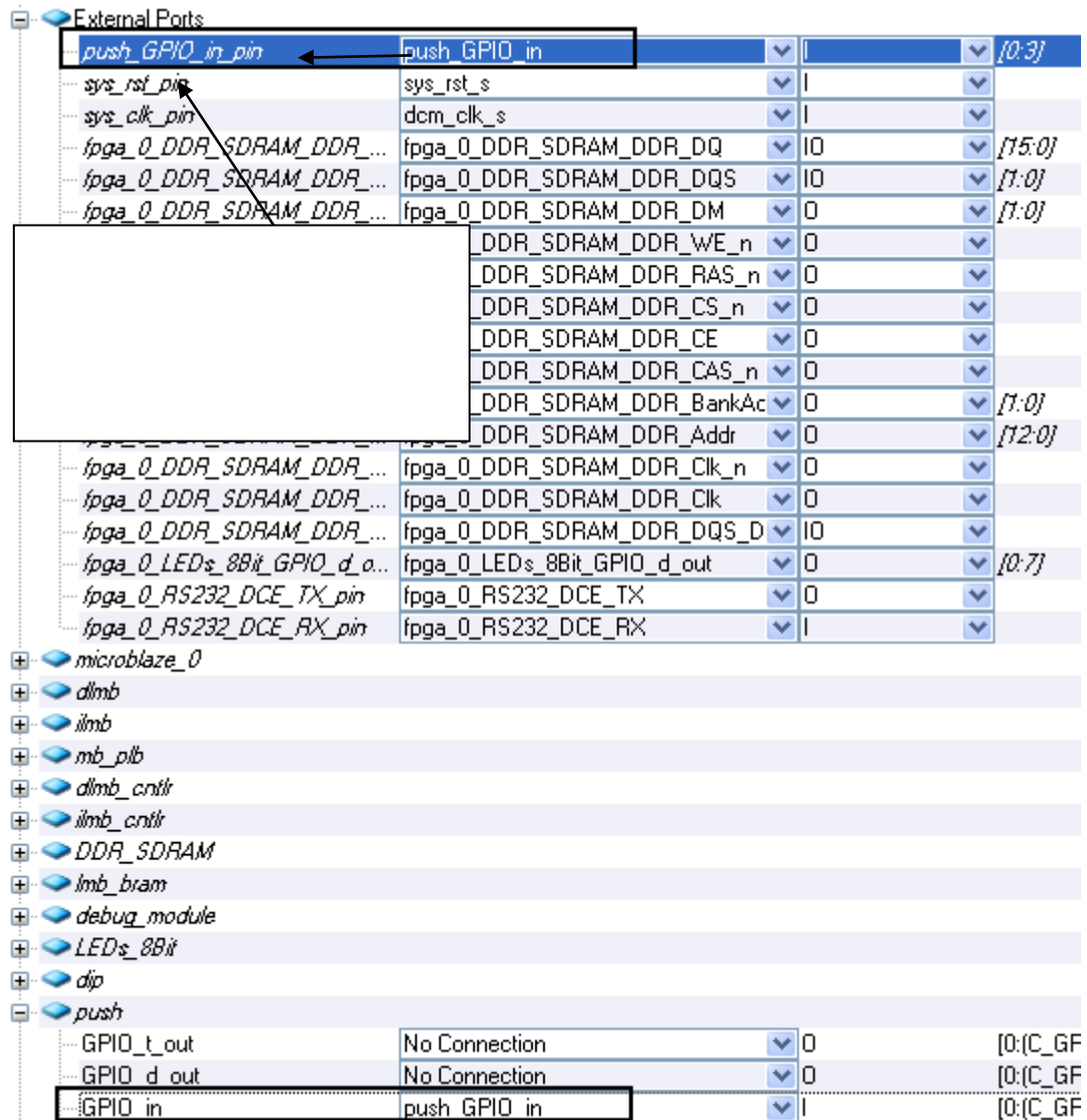


Figure 1-28. GPIO_in Port Connection Added to push Instance

- Set the GPIO_in port of dip as external.

The GPIO_in ports of both dip and push are now connected externally on the FPGA (Figure 1-29).

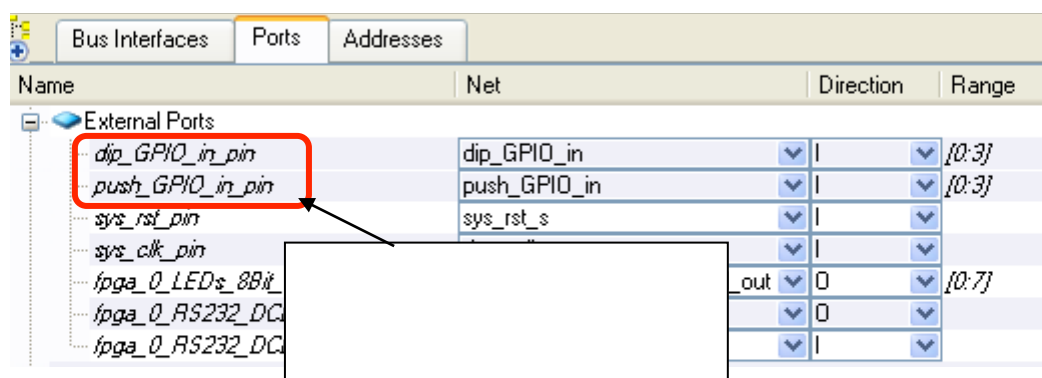


Figure 1-29. Push and DIP Instances External Ports

- ④ Click on the **system.ucf** file under the **Project** tab and add the following code to assign pins to push buttons (lab4_1.ucf).

```

613
614 ##### Pin location constraints for the push buttons
615 NET push_GPIO_in_pin<0> LOC=V4 | IOSTANDARD = LVTTTL | PULLDOWN; # North Push Button
616 NET push_GPIO_in_pin<1> LOC=H13 | IOSTANDARD = LVTTTL | PULLDOWN; # East Push Button
617 NET push_GPIO_in_pin<2> LOC=D18 | IOSTANDARD = LVTTTL | PULLDOWN; # West Push Button
618 Net push_GPIO_in_pin<3> LOC=V16 | IOSTANDARD = LVTTTL | PULLDOWN; # Center Push Button
619
620 ##### Pin location constraints for the DIP switches
621 NET dip_GPIO_in_pin<0> LOC=L13 | IOSTANDARD = LVTTTL | PULLUP; # Switch0
622 NET dip_GPIO_in_pin<1> LOC=L14 | IOSTANDARD = LVTTTL | PULLUP; # Switch1
623 NET dip_GPIO_in_pin<2> LOC=H18 | IOSTANDARD = LVTTTL | PULLUP; # Switch2
624 Net dip_GPIO_in_pin<3> LOC=N17 | IOSTANDARD = LVTTTL | PULLUP; # Switch3
625

```

Figure 1-30. UCF file (pin assignments).

- ⑤ Save the system.ucf and close it

Add Software Application and Compile

Step 8



Edit the existing c program to implement the functionality of push button and LEDs. Compile the program.

- ② Click on **Applications** tab and under **Sources**, edit the **TestApp_Memory.c** file. A snippet of the source code is shown in **Figure 1-31**. This source code is defined from lab4_1.c.

```

1  #include "xparameters.h"
2  #include "xgpio.h"
3  #include "xutil.h"
4
5
6
7  //=====
8
9  int main (void)
10 {
11
12     XGpio dip, push;
13     int i, psb_check, dip_check;
14
15     //xil_printf("-- Start of the Program --\r\n");
16
17     XGpio_Initialize(&dip, XPAR_DIP_DEVICE_ID);
18     XGpio_SetDataDirection(&dip, 1, 0xffffffff);
19
20     XGpio_Initialize(&push, XPAR_PUSH_DEVICE_ID);
21     XGpio_SetDataDirection(&push, 1, 0xffffffff);
22
23     while (1)
24     {
25         psb_check = XGpio_DiscreteRead(&push, 1);
26         xil_printf("Push Buttons Status %x\r\n", psb_check);
27         dip_check = XGpio_DiscreteRead(&dip, 1);
28         xil_printf("DIP Switch Status %x\r\n", dip_check);
29
30         for (i=0; i<999999; i++);
31     }
32
33 }

```

Figure 1-31. Snippet of source code.

- ③ In the Application tab, double-click on compiler options to open the **Compiler Options** dialogue box.
- ④ In the **Environment** tab, select the option **Use Default Linker Script**.

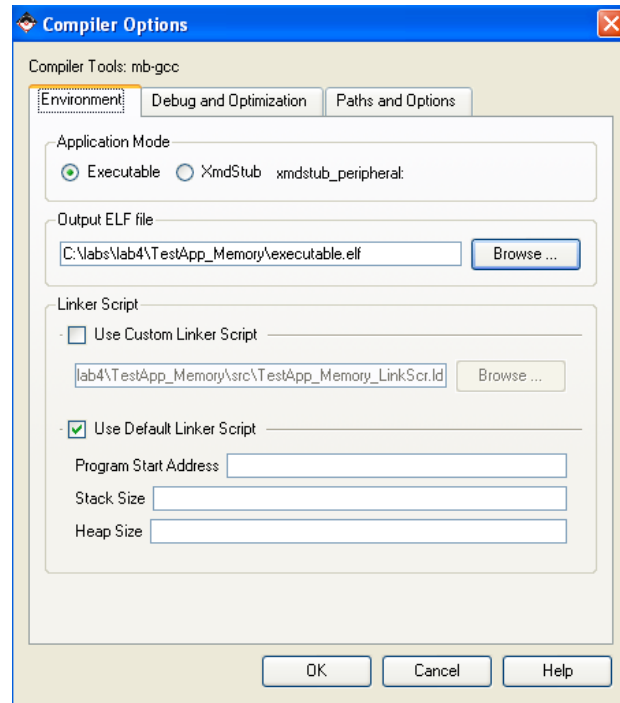


Figure 1-32. Setting the Default Linker Script

- ⑤ In the **Debug and Optimization** tab, set the optimization to **No Optimization**.

This will ensure that the **for loop** (used for software delay) in the source code is not optimized away.

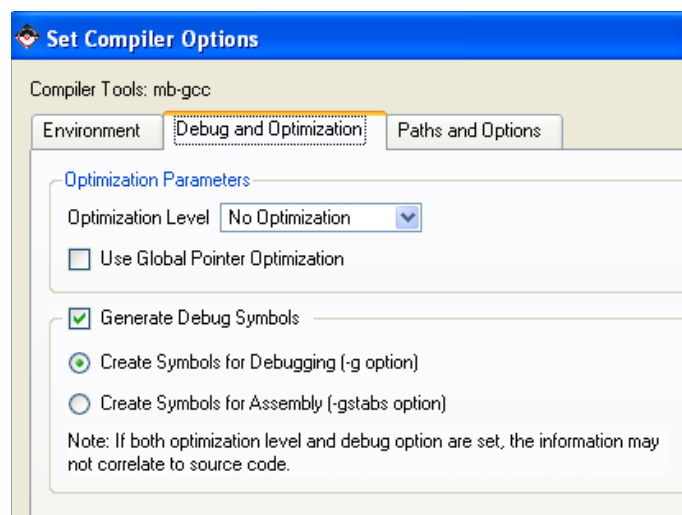



Figure 2-13. Setting the Optimization level

- ⑤ Click on  to compile the source code. Make sure that it compiles error free

Note: This will automatically run LibGen to generate the required libraries if it has not been done already.

Verify the Design in Hardware

Step 9



Download the bitstream to the Spartan-3E xc3s500e device.

❶ Start a HyperTerminal session

- Baud rate: 115200
- Data bits: 8
- Parity: none
- Stop bits: 1
- Flow control: none

❷ Connect and power up the Spartan-3E starter kit.

❸ Select **Device Configuration → Update Bitstream**

This may take a few minutes to synthesize, implement, and generate the bitstream.

❹ Download the bitstream by selecting **Device Configuration → Download Bitstream**

Note: Once the bitstream is downloaded, you should see the DONE LED ON and a message displayed in HyperTerminal as shown in **Figure 1-34**

```
DIP Switch Status 0
Push Buttons Status 0
DIP Switch Status 0
Push Buttons Status 0
DIP Switch Status 0
Push Buttons Status 0
DIP Switch Status 0
```

Figure 1-34. Screen Shot after the BitStream Downloading

❺ After pressing the **buttons** and toggling the **switches**, and you should see the corresponding values being displayed on the HyperTerminal (**Figure 1-35**)

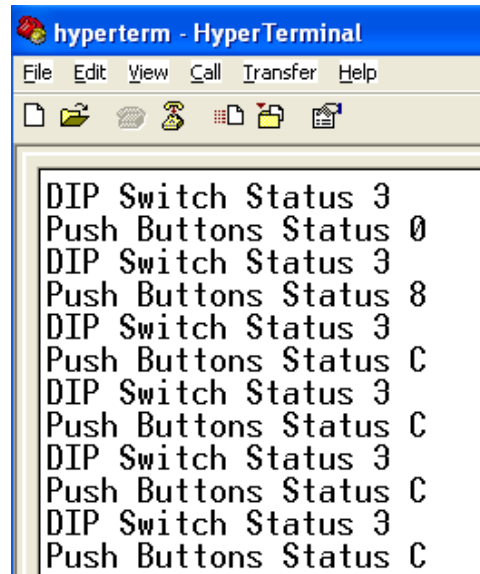


Figure 1-35. Push button and DIP switch status displayed on hyperterminal

- ⑥ Disconnect and close the HyperTerminal window, and also close XPS

Adding Custom IP to an Embedded System

Step 9

You will extend the hardware design by creating and adding a PLB peripheral (refer to MYIP in Figure 1-36) to the system, and connecting it to the LCD on the Spartan-3E kit. You will use the Create and Import Peripheral Wizard of Xilinx Platform Studio (XPS) to generate the peripheral templates. You will complete the peripheral by adding LCD interface logic in the templates. Next, you will connect the peripheral to the system and add pin location constraints to connect the LCD controller peripheral to the on-board LCD. Finally, you will verify operation in hardware using the provided software application.

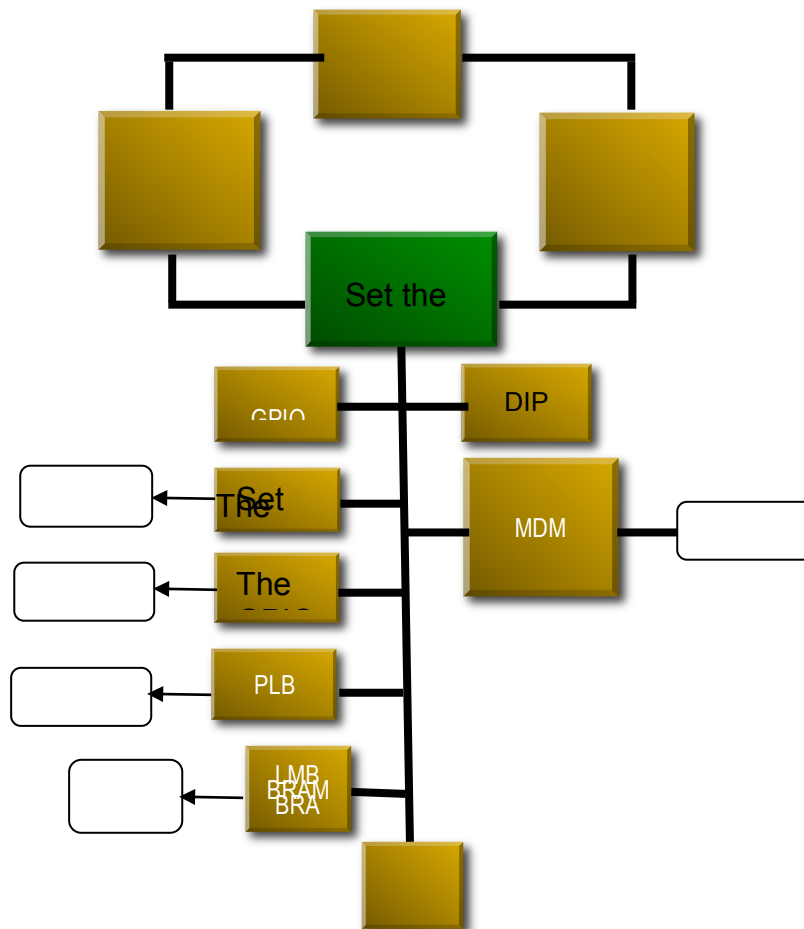


Figure 1-36. Design updated from previous lab

Generate a Peripheral Template

Step 10



You will use the Create/Import Peripheral Wizard to create a PLB bus peripheral template.

- ❶ In XPS, select **Hardware** → **Create or Import Peripheral** to start the wizard
- ❷ Click **Next** to continue to the Create and Import Peripheral Wizard flow selection (**Figure 1-36**).

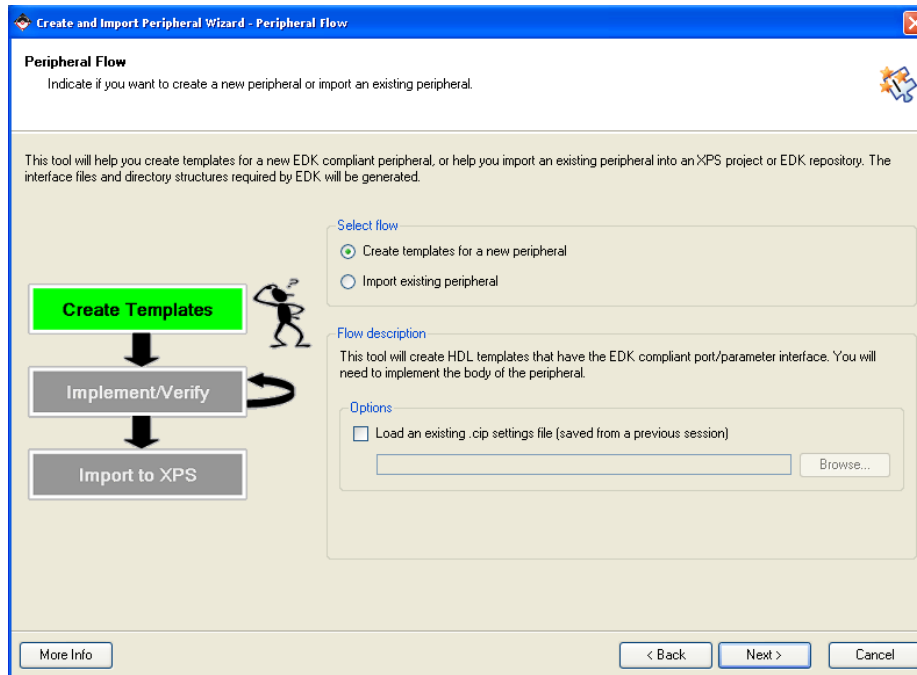


Figure 1-36. Create and Import User Peripheral Dialog Box

- ③ In the **Select Flow** panel, select **Create templates for a new peripheral** and click **Next**
- ④ Click **next** with the default option **To an XPS project** selected.

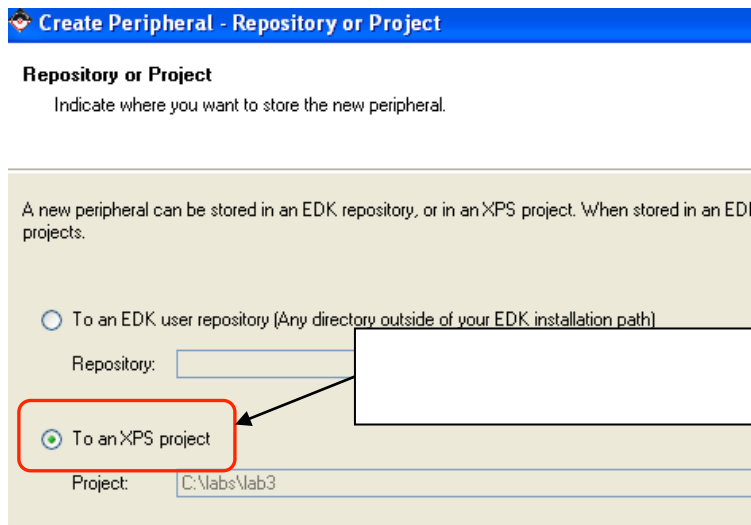


Figure 1-37. Repository or Project Dialog Box

- ⑤ Click **Next** and enter **lcd_ip** in the Name field, leave the default version number of 1.00.a, click **Next** (see **Figure 1-38**)

Create Peripheral - Name and Version

Name and Version
Indicate the name and version of your peripheral.

Enter the name of your peripheral. This name will be used as the top HDL design entity.

Name:

Version: 1.00.a

Major revision: Minor revision: Hardware/Software compatibility revision:

Figure 3-4. Provide Core Name and Version Number

- ⑥ Select **Processor Local Bus (PB v4.6)**, and click **Next**

Create Peripheral - Bus Interface

Bus Interface
Indicate the bus interface supported by your peripheral.

To which bus will this peripheral be attached?

☒ Processor Local Bus (PLB v4.6)

☐ Fast Simplex Link (FSL)

ATTENTION
Refer to the following documents to get a better understanding of how user peripherals connect to the CoreConnect(TM) buses (including PLB v4.6 interconnect and OPB/PLB v3.4 interconnect) and the FSL interface.

NOTE - Select the bus interface above and the corresponding link(s) will appear below for that interface.

[CoreConnect Specification](#)
[PLB \(v4.6\) Slave IPIF Specification for single data beat transfer](#)
[PLB \(v4.6\) Slave IPIF Specification for burst data transfer](#)
[PLB \(v4.6\) Master IPIF Specification for single data beat transfer](#)
[PLB \(v4.6\) Master IPIF Specification for burst data transfer](#)

Note
Xilinx recommends using the new PLB v4.6 bus standard, however, the wizard still supports the OPB and PLB v3.4 bus interfaces.

☐ Enable OPB and PLB v3.4 bus interfaces

Figure 1-39. Select the PLB bus



Continuing with the wizard, select **User Logic S/W Register support**. Select only one software accessible register of 32-bit width. Generate template driver files.

In the **IPIF Services** panel, deselect **Include data phase timer** and click **Next**

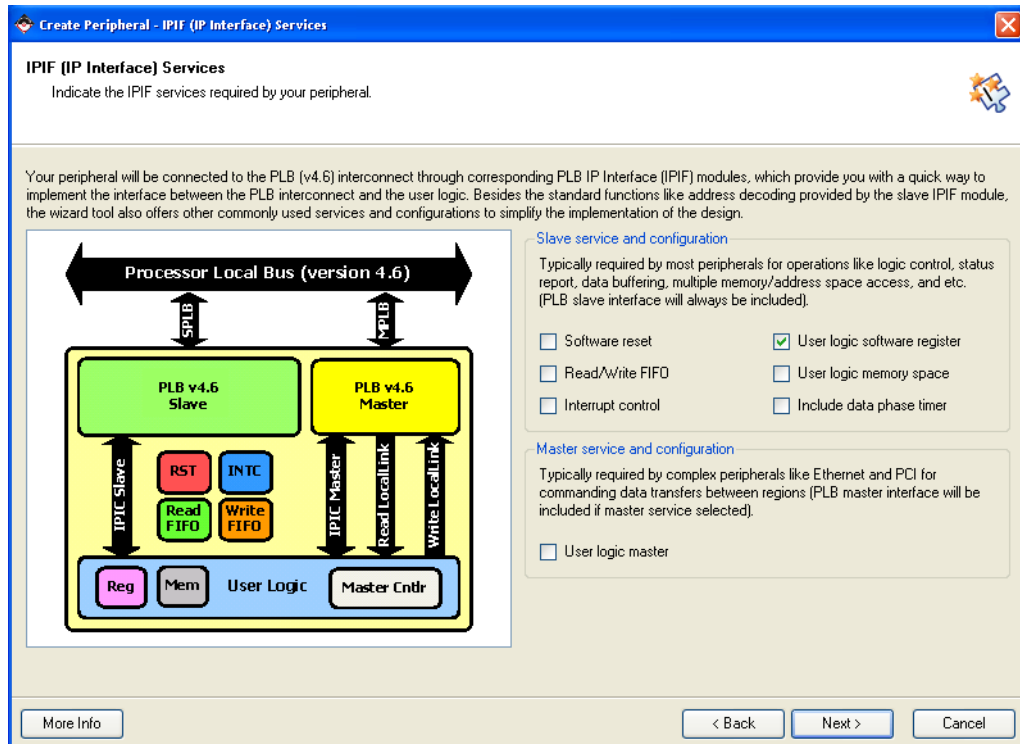


Figure 1-40. IPIF Services Dialog Box

- Click **Next**, accepting the default data width, and no burst and cache line support. Click **Next** to accept default number of registers (one)

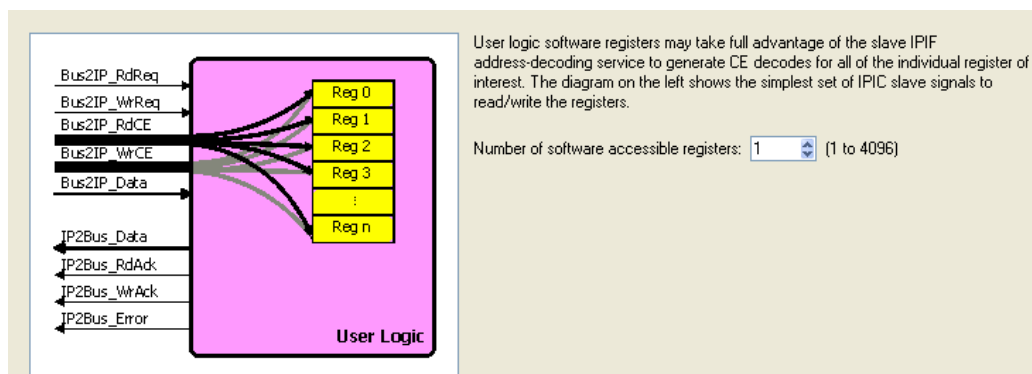


Figure 1-41. User SW Registers

- Scroll through the **IP Interconnect (IPIC)** panel, which displays the default IPIC signals that are available for the user logic based on the previous selection. Click **Next**

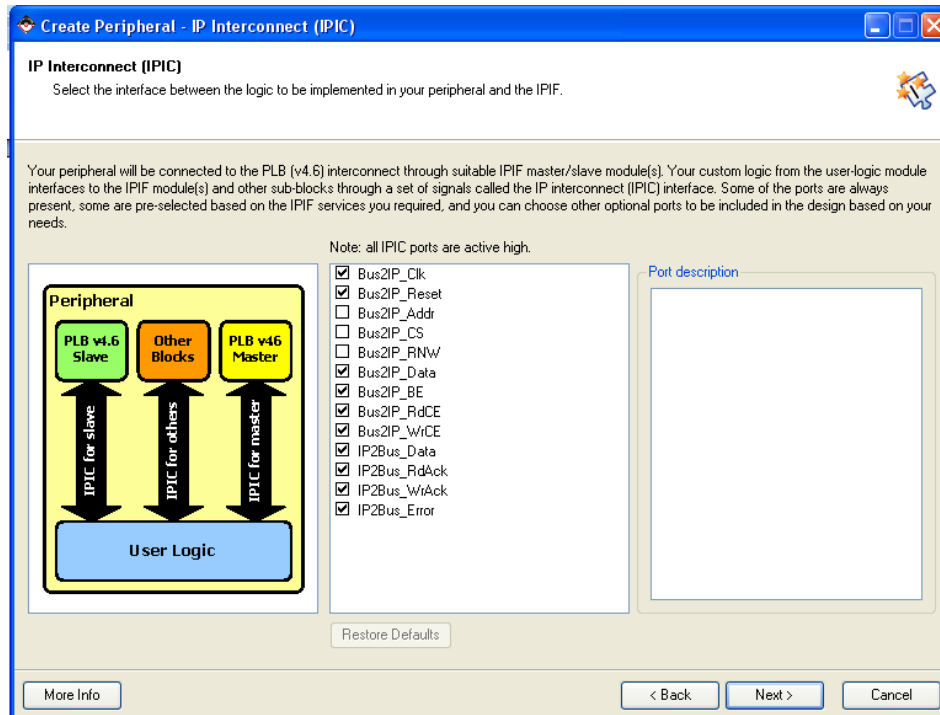


Figure 1-42. IP Interconnect (IPIC) Dialog Box

- ④ In the **Peripheral Simulation Support** panel, leave **Generate BFM simulation platform** unchecked, and click **Next**

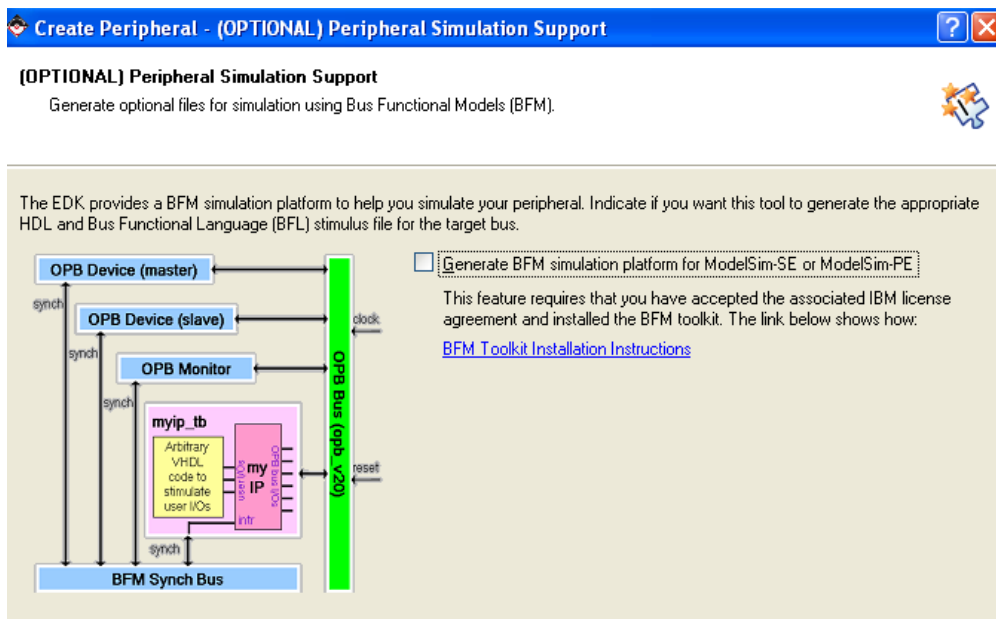


Figure 1-43. Peripheral Simulation Support Dialog Box

- ⑤ In the **Peripheral Implementation Options** panel, click **Generate template driver files to help you to implement software interface**, leaving others unchecked

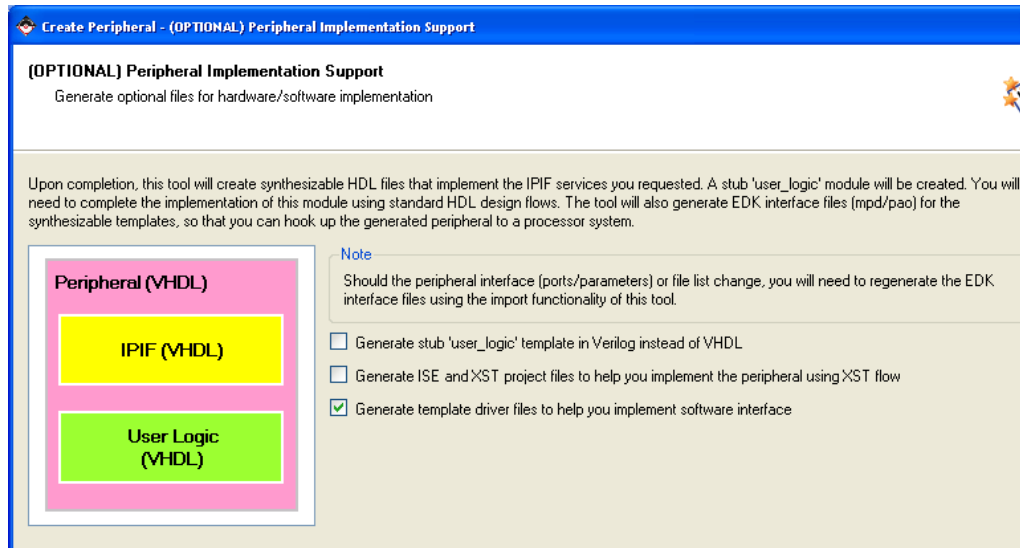


Figure 1-44. Peripheral Implementation Options Dialog Box

- ⑥ Click **Next**, and you will see the summary information panel

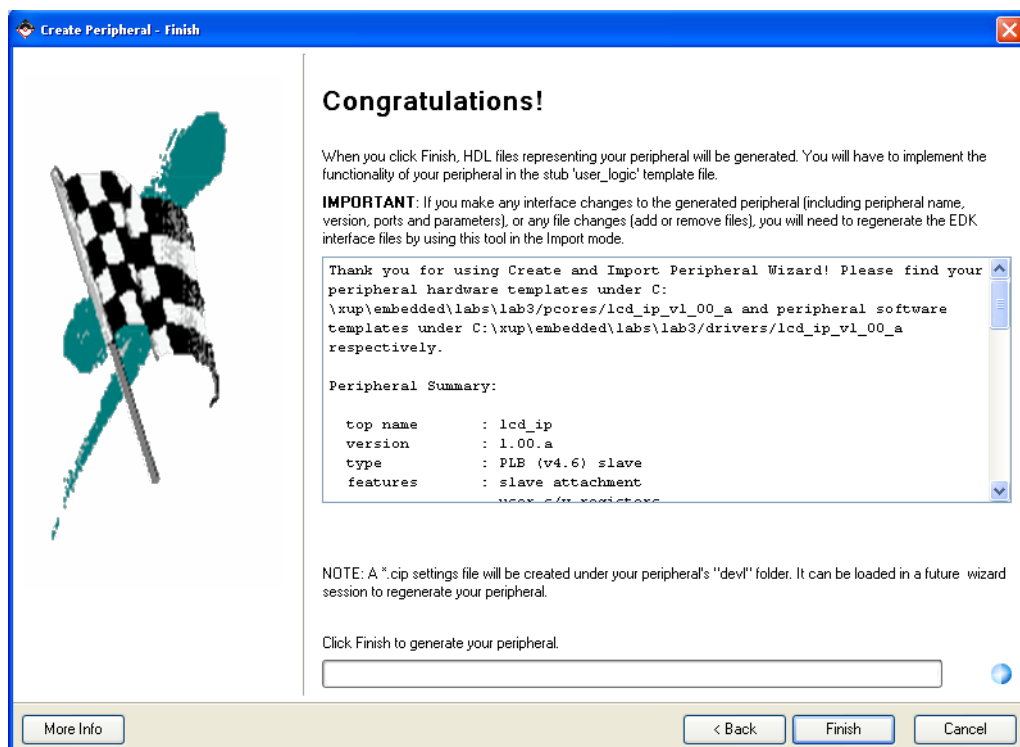


Figure 1-45. Congratulations Dialog Box

- ⑦ Click **Finish** to close the wizard
- ⑧ Click on **IP Catalog** tab in XPS and observe that **lcd_ip** is added to the **Project Local pcores** repository (**Figure 1-46**).

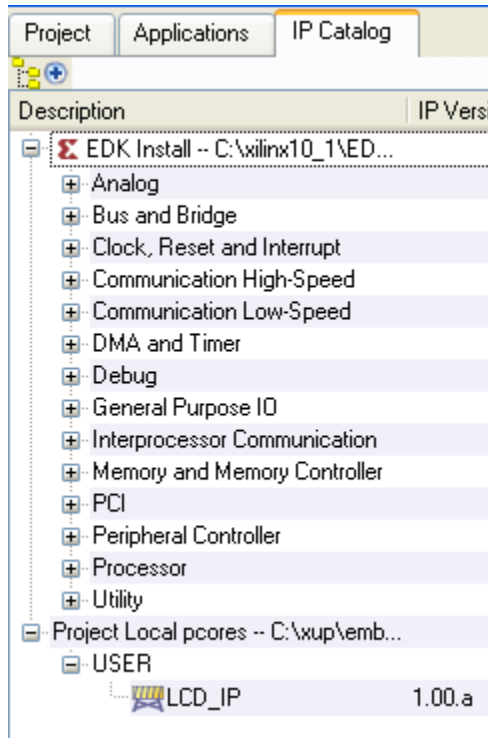


Figure 1-46. IP Catalog Updated Entry

The peripheral which you just added becomes part of the available cores list. Use Windows Explorer to browse to your project directory and ensure that the following structure has been created by the Create and Import Peripheral Wizard (**Figure 1-47**)

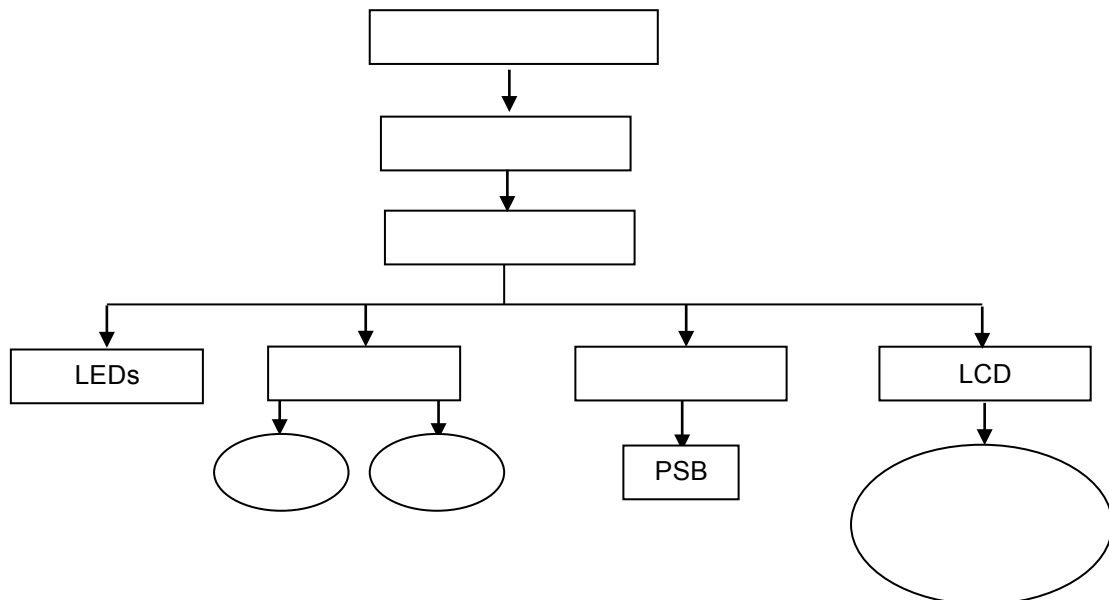


Figure 1-47. Structure Created by the Create and Import Peripheral Wizard

Create the Peripheral

Step 11



Update the MPD file to include the **lcd** data output of the LCD controller peripheral so the port can be connected in XPS.

Add a port called “lcd” to the MPD file.

- ❶ Open **lcd_ip_v2_1_0.mpd** in the **pcores\lcd_ip_v1_00_a\data** under **lab4** directory.
- ❷ Add following line before the **SPLB_Clk** port under the **Ports** section

PORT lcd = “”, DIR = O, VEC = [0:6]

```
PARAMETER C_SPLB_NATIVE_DWIDTH = 32, DT = INTEGER, BUS = SPLB,
PARAMETER C_SPLB_P2P = 0, DT = INTEGER, BUS = SPLB, RANGE = {0,
PARAMETER C_SPLB_SUPPORT_BURSTS = 0, DT = INTEGER, BUS = SPLB,
PARAMETER C_SPLB_SMALLEST_MASTER = 32, DT = INTEGER, BUS = SPLB,
PARAMETER C_SPLB_CLK_PERIOD_PS = 10000, DT = INTEGER, BUS = SPLB,
PARAMETER C_FAMILY = virtex5, DT = STRING
```

Ports

PORT lcd = “”, DIR = O, VEC = [0:6]

```
PORT SPLB_Clk = “”, DIR = I, SIGIS = CLK, BUS = SPLB
PORT SPLB_Rst = SPLB_Rst, DIR = I, SIGIS = RST, BUS = SPLB
PORT PLB_ABus = PLB_ABus, DIR = I, VEC = [0:31], BUS = SPLB
PORT PLB_UABus = PLB_UABus, DIR = I, VEC = [0:31], BUS = SPLB
PORT PLB_PValid = PLB_PValid, DIR = I, BUS = SPLB
PORT PLB_SValid = PLB_SValid, DIR = I, BUS = SPLB
```

Figure 1-48. Update the MPD file for the LCD Controller Peripheral

- ❸ Save the file and close



Create the LCD controller using the appropriate HDL template files generated from the Create/Import peripheral wizard: **lcd_ip.vhd** and **user_logic.vhd**. You can edit these files using a standard text editor.

- ❶ Open **lcd_ip.vhd** in the **pcores\lcd_ip_v1_00_a\hdl\vhdl** directory.
- ❷ Add user port **lcd** of width 7 under **USER ports added here** token (see Figure 1-49)

```

148 C_SPLB_AWIDTH      : integer      := 32;
149 C_SPLB_DWIDTH      : integer      := 128;
150 C_SPLB_NUM_MASTERS  : integer      := 8;
151 C_SPLB_MID_WIDTH    : integer      := 3;
152 C_SPLB_NATIVE_DWIDTH : integer    := 32;
153 C_SPLB_P2P          : integer      := 0;
154 C_SPLB_SUPPORT_BURSTS : integer    := 0;
155 C_SPLB_SMALLEST_MASTER : integer   := 32;
156 C_SPLB_CLK_PERIOD_PS : integer     := 10000;
157 C_FAMILY            : string        := "virtex5"
158 -- DO NOT EDIT ABOVE THIS LINE -----
159 );
160 port
161 (
162     -- ADD USER PORTS BELOW THIS LINE -----
163     --USER ports added here
164     lcd : out std_logic_vector(0 to 6);
165     -- ADD USER PORTS ABOVE THIS LINE -----
166
167     -- DO NOT EDIT BELOW THIS LINE -----
168     -- Bus protocol ports, do not add to or delete
169     SPLB_Clk      : in  std_logic;
170     SPLB_Rst      : in  std_logic;
171     PLB_ABus      : in  std_logic_vector(0 to 31);
172     PLB_UABus     : in  std_logic_vector(0 to 31);

```

Figure 1-49. Add the User Port LCD

- ④ Search for next `--USER` and add port mapping statement, save the file and then close it

```

373 -----
374 -- instantiate User Logic
375 -----
376 USER_LOGIC_I : entity lcd_ip_v1_00_a.user_logic
377 generic map
378 (
379     -- MAP USER GENERICS BELOW THIS LINE -----
380     --USER generics mapped here
381     -- MAP USER GENERICS ABOVE THIS LINE -----
382
383     C_SLV_DWIDTH      => USER_SLV_DWIDTH,
384     C_NUM_REG         => USER_NUM_REG
385 )
386 port map
387 (
388     -- MAP USER PORTS BELOW THIS LINE -----
389     --USER ports mapped here
390     lcd                => lcd,
391     -- MAP USER PORTS ABOVE THIS LINE -----
392
393     Bus2IP_Clk        => ipif_Bus2IP_Clk,
394     Bus2IP_Reset      => ipif_Bus2IP_Reset,
395     Bus2IP_Data       => ipif_Bus2IP_Data,

```

Figure 1-50. Add Port Mapping Statement

- ⑤ Open `user_logic.vhd` file from the `vhdl` directory and add `lcd` port definition in the USER Ports area


```

84 entity user_logic is
85   generic
86   (
87     -- ADD USER GENERICS BELOW THIS LINE -----
88     --USER generics added here
89     -- ADD USER GENERICS ABOVE THIS LINE -----
90
91     -- DO NOT EDIT BELOW THIS LINE -----
92     -- Bus protocol parameters, do not add to or delete
93     C_DWIDTH                : integer          := 32;
94     C_NUM_CE                : integer          := 1
95     -- DO NOT EDIT ABOVE THIS LINE -----
96   );
97   port
98   (
99     -- ADD USER PORTS BELOW THIS LINE -----
100    --USER ports added here
101    lcd : out std_logic_vector (0 to 6);
102    -- ADD USER PORTS ABOVE THIS LINE -----
103

```

Figure 1-50. Add the lcd Port Definition

- ⑥ Search for next **--USER** and then enter the internal signal declaration according to the figure below

```

119 attribute SIGIS : string;
120 attribute SIGIS of Bus2IP_Clk    : signal is "CLK";
121 attribute SIGIS of Bus2IP_Reset  : signal is "RST";
122
123 end entity user_logic;
124
125 -----
126 -- Architecture section
127 -----
128
129 architecture IMP of user_logic is
130
131   --USER signal declarations added here, as needed for user logic
132   signal lcd_i                : std_logic_vector(0 to 6);
133
134   -----
135   -- Signals for user logic slave model s/w accessible register example
136   -----
137   signal slv_reg0              : std_logic_vector(0 to C_SLV_DWIDTH-1);
138   signal slv_reg_write_sel     : std_logic_vector(0 to 0);
139   signal slv_reg_read_sel      : std_logic_vector(0 to 0);
140   signal slv_ip2bus_data       : std_logic_vector(0 to C_SLV_DWIDTH-1);
141   signal slv_read_ack          : std_logic;
142   signal slv_write_ack         : std_logic;

```

Figure 1-50. Internal Signal Declaration for the User Logic

- ⑦ Search for **--USER logic implementation** and add the following code or copy it from lab_4_2_lcd_user_logic.vhd.

```

144 begin
145
146 --USER logic implementation added here
147 lcd_PROC : process (Bus2IP_Clk) is
148 begin
149     if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
150         if Bus2IP_Reset = '1' then
151             lcd_i <= (others => '0');
152         else
153             if Bus2IP_WrCE(0) = '1' then
154                 lcd_i <= Bus2IP_Data(25 to 31);
155             end if;
156         end if;
157     end if;
158 end process lcd_PROC;
159 lcd <= lcd_i;
160

```

Figure 1-51. Add Code

- ③ Save changes and close the **user_logic.vhd**
- ④ Select **Project** → **Rescan User Repositories** to have the changes in effect

Add and Connect the Peripheral

Step 12



Add and connect the LCD peripheral to the PLB bus in the System Assembly View. Make internal and external port connections. Assign an address range to it. Establish the LCD data port as external FPGA pins and assign the pin location constraints so the peripheral interfaces to the LCD display on the Spartan-3E starter kit.

- ① In **IP Catalog**, select **lcd_ip** core, drag and drop it in the **System Assembly View** panel
- ② Make sure that the **Bus Interfaces** filter is selected in the System Assembly View and click on the circle in the bus connection diagram to make bus connection (**Figure 1-52**)

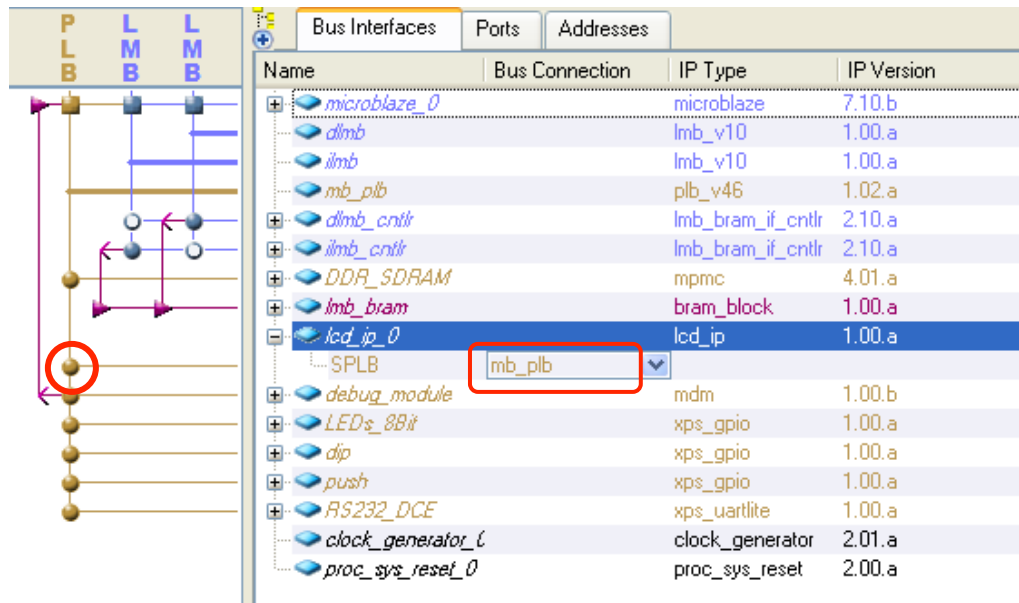


Figure 1-52. Making Bus Connection

- Select the **Ports** filter, and connect the **lcd** port of the **lcd_ip_0** instance as an external pin by selecting **Make External** (Figure 3-21)

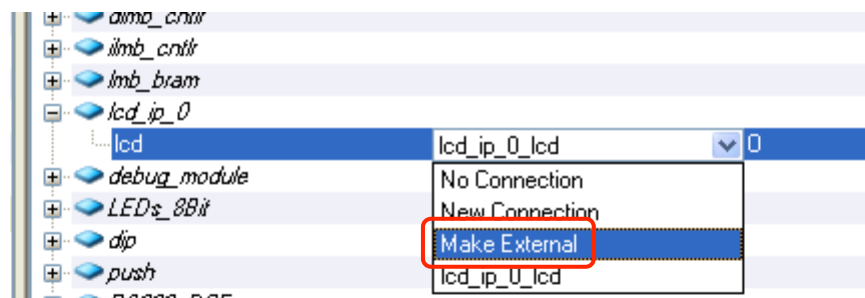


Figure 1-53. Assign the lcd_0 Instance

- Select **Addresses** filter and lock addresses of all devices except for the **lcd_ip_0** instance.
- Change the size of the **lcd** peripheral to **64K** and click the **Generate Addresses** button.

Your results should look similar to that below (as shown in **Figure 1-54**)

Bus Interfaces Ports Addresses						
Instance	Name	Base Address	High Address	Size	Bus Interface(s)	
lcd_ip_0	C_BASEADDR	0xc400000	0xc40ffff	64K	SPLB	
dlmb_cntrlr	C_BASEADDR	0x00000000	0x00001fff	8K	SLMB	
ilmb_cntrlr	C_BASEADDR	0x00000000	0x00001fff	8K	SLMB	
debug_module	C_BASEADDR	0x84400000	0x8440ffff	64K	SPLB	
push	C_BASEADDR	0x81400000	0x8140ffff	64K	SPLB	
dip	C_BASEADDR	0x81420000	0x8142ffff	64K	SPLB	
LEDs_8Bit	C_BASEADDR	0x81440000	0x8144ffff	64K	SPLB	
RS232_DCE	C_BASEADDR	0x84000000	0x8400ffff	64K	SPLB	
DDR_SDRAM	C_MPMC_BASEADDR	0x8c000000	0x8fffffff	64M	SPLB0	

Figure 1-54. Generate Addresses



Modify the system.ucf file to assign external LCD controller connections to the proper FPGA pin locations.

- ❶ Open the **system.ucf** file by double-clicking the **UCF File: data\system.ucf** entry under Project Files in the System tab. Copy it from lab_4_2_lcd.ucf

```
#-----
# IO Pad Location Constraints / Properties for Character LCD GPIO
#-----
NET lcd_ip_0_lcd_pin<0> LOC = M18 | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW; ;
NET lcd_ip_0_lcd_pin<1> LOC = L18 | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW; ;
NET lcd_ip_0_lcd_pin<2> LOC = L17 | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW; ;
NET lcd_ip_0_lcd_pin<3> LOC = M15 | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW; ;
NET lcd_ip_0_lcd_pin<4> LOC = P17 | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW; ;
NET lcd_ip_0_lcd_pin<5> LOC = R16 | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW; ;
NET lcd_ip_0_lcd_pin<6> LOC = R15 | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW; ;
```

Figure 1-55. Adding UCF Constraints

- ❷ Save and close the file

Verify the Design in Hardware

Step 12



Add a software new software program. Use EDK to generate the configuration file and program the Spartan-3E xc3s500e-4fg320 device.

- ❶ Click on the **Applications** tab and remove **TestApp_Memory.c** file from the sources
- ❷ Add **lcd.c** file in sources
- ❸ Connect the USB and RS-232 cables to the XUP Spartan-3E board and power it up.
- ❹ Start a HyperTerminal with the following settings
 - Baud rate: 115200
 - Data bits: 8
 - Parity: none
 - Stop bits: 1
 - Flow control: none
- ❺ From EDK, click on **Device Configuration** → **Download Bitstream** to download the system to the FPGA

Note: this will perform the following steps

- Run platgen to generate the netlists
- Generate the bitstream
- Run libgen to generate the libraries and drivers
- Compile the program to generate the executable
- Update the BRAMs in the bitstream with the executable
- Download the bitstream to the FPGA

Note: Once the bitstream is downloaded, you should see the DONE LED ON and a message displayed in HyperTerminal as shown in **Figure 1-55**

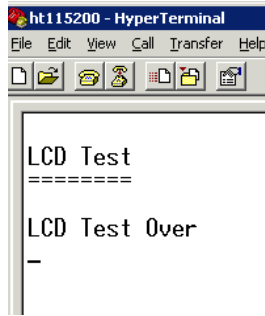


Figure 1-55. Screen Shot after the BitStream Downloading

You should see LCD Test Over in the HyperTerminal window and “MicroBlaze and FPGAs rules” on the LCD panel on the XUP Spartan-3E board

Update a Basic Software Application

Step 13



Run LibGen to generate `xparameters.h` file which defines various symbolic parameters. Modify a software program to display the DIP switch settings on the LEDs.

- ❶ Run libgen by selecting **Software → Generate Libraries and BSPs** to generate `xparameters.h` file
 - LibGen writes the `xparameters.h` file, which provides critical information for driver function calls.
- ❷ In the Applications tab, remove `lcd.c` and add `lab4_1.c` to the **TestApp_Memory** project

You will extend the functionality in `lab4_1.c` by adding code to display switch settings on the LEDs.
- ❸ Open the GPIO API documentation by right-clicking on **LEDs_8Bit** peripheral in the System Assembly View and selecting **Driver: gpio_v2_12_a → View API Documentation**
- ❹ View the various C and Header files associated with the GPIO by selecting **File List** at the top of the page.
- ❺ Click the header file `xgpio.h` and review the list of available function calls for the GPIO
- ❻ The following steps must be performed in the software application to enable writing to the GPIO: **1) Initialize the GPIO, 2) Set data direction, and 3) Write the data**

Find the descriptions for the following functions by clicking links:

XGpio_Initialize (XGpio *InstancePtr, ul6 DeviceId)

- **InstancePtr** is a pointer to an Xgpio instance. The memory the pointer references must be pre-allocated by the caller. Further calls to manipulate the component through the XGpio API must be made with this pointer.
- **DeviceId** is the unique ID of the device controlled by this XGpio component. Passing in a device ID associates the generic XGpio instance to a specific device, as chosen by the caller or application developer.

```
XGpio_SetDataDirection (XGpio * InstancePtr, unsigned
                        Channel, u32 DirectionMask)
```

- **InstancePtr** is a pointer to the XGpio instance to be worked on.
- **Channel** indicates the channel of the GPIO (1 or 2) to operate on
- **DirectionMask** is a bit mask specifying which discretes are input and which are output. Bits set to 0 are output and bits set to 1 are input.

```
XGpio_DiscreteWrite (XGpio *InstancePtr, unsigned channel,
                    u32 data)
```

- **InstancePtr** is a pointer to the XGpio instance to be worked on.
- **Channel** indicates the channel of the GPIO (1 or 2) to operate on
- **Data** is the data written to the discrete registers.

- 7 Open the header file **xparameters.h** by double-clicking on **Generated Header: microblaze_0/include/xparameters.h** under the microblaze_0 instance for the **TestApp_Memory** project in the Applications tab.

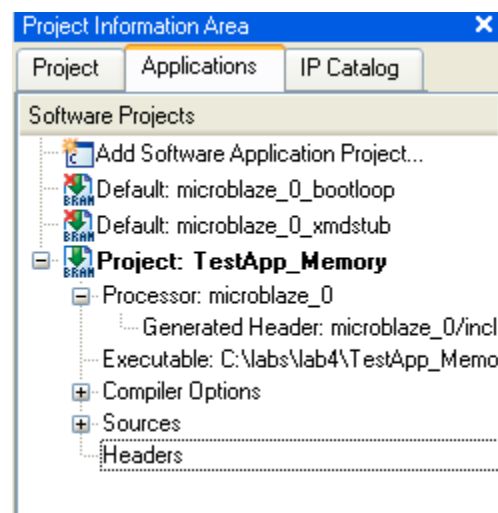


Figure 1-56. Double-Click the Generated Header File

- In the **xparameters.h** file, find the following **#define** used to identify the **LEDs_8Bit** peripheral:

```
#define XPAR_LEDS_8BIT_DEVICE_ID
```

Note: The **LEDs_8BIT** matches the instance name assigned in the MHS file for this peripheral.

This **#define** can be used in the **XGpio_Init** function call.

- 8 Modify your C code to echo the dip switch settings on the **LEDs** (**Figure 1-57**) and save the application.

```

1  #include "xparameters.h"
2  #include "xgpio.h"
3  #include "xutil.h"
4
5  //=====
6
7  int main (void)
8  {
9
10     XGpio dip, push;
11     int i, psb_check, dip_check;
12     // define instance pointer for LEDs_8Bit device
13     XGpio LEDs8_Bit;
14
15     xil_printf("-- Start of the Program --\r\n");
16
17     XGpio_Initialize(&dip, XPAR_DIP_DEVICE_ID);
18     XGpio_SetDataDirection(&dip, 1, 0xffffffff);
19
20     XGpio_Initialize(&push, XPAR_PUSH_DEVICE_ID);
21     XGpio_SetDataDirection(&push, 1, 0xffffffff);
22
23     // initialize and set data direction for LEDs 8Bit device
24     XGpio_Initialize(&LEDs8_Bit, XPAR_LEDS_8BIT_DEVICE_ID);
25     XGpio_SetDataDirection(&LEDs8_Bit, 1, 0x0);
26
27     while (1)
28     {
29         psb_check = XGpio_DiscreteRead(&push, 1);
30         xil_printf("Push Buttons Status %x\r\n", psb_check);
31         dip_check = XGpio_DiscreteRead(&dip, 1);
32         xil_printf("DIP Switch Status %x\r\n", dip_check);
33
34         // output dip switches value on LEDs 8Bit device
35         XGpio_DiscreteWrite(&LEDs8_Bit, 1, dip_check);
36
37         for (i=0; i<999999; i++);
38     }
39 }
40

```

Figure 1-57. Partially Completed C File

- ⑨ Click the compile button  to compile the program.

SDK Design

Step 12

This lab guides you through the process of adding timers to an embedded system and writing a software application that utilizes these timers. The Software Developers Kit (SDK) will be used to create and debug the software application.

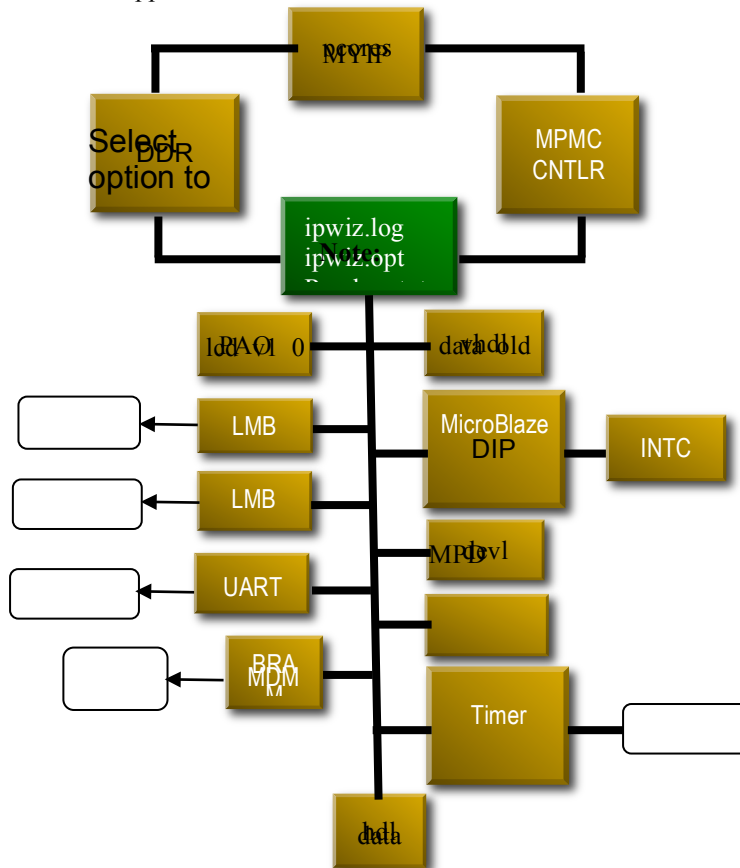


Figure 1-58. Design Updated from Previous step

Add a Timer and Interrupt Controller

Step 13



Add the XPS timer and XPS Interrupt Controller peripherals to the design from the IP Catalog, and connect them to the system according to the following table.

xps_intc_0 instance	
plb_clk	sys_clk_s
Intr	timer1
Irq	Microblaze_0_INTERRUPT
delay instance	
CaptureTrig0	net_gnd
Interrupt	timer1
microblaze_0 instance	
INTERRUPT	Microblaze_0_INTERRUPT

- ❶ Add the **XPS Timer/Counter** peripheral from the **DMA and Timer** section of the IP Catalog and change its instance name to **delay**
- ❷ Add the **XPS Interrupt Controller** peripheral from the **Clock, Reset, and Interrupt** section of the IP Catalog
- ❸ Connect the **timer** and **interrupt controller** as a 's' (slave) device to the PLB bus (see **Figure 1-59**)

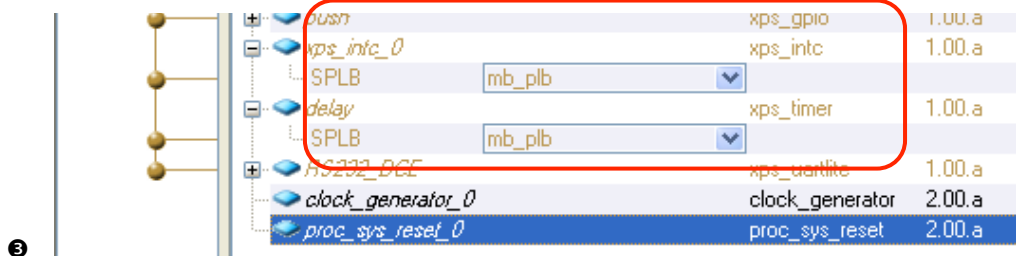


Figure 1-59. Add and Connect the Interrupt Controller and Timer Peripherals

- ❹ Select **size** as **64K bytes** from drop down box and click **Generate Addresses**.

Your results should look similar to that indicated in the figure below.


Bus Interfaces					
Ports					
Addresses					
Instance	Name	Base Address	High Address	Size	Bus Interface(s)
lcd_ip_0	C_BASEADDR	0xc400000	0xc40ffff	64K	SPLB
dlmb_cntlr	C_BASEADDR	0x00000000	0x00001fff	8K	SLMB
ilmb_cntlr	C_BASEADDR	0x00000000	0x00001fff	8K	SLMB
debug_module	C_BASEADDR	0x84400000	0x8440ffff	64K	SPLB
xps_bram_if_cntlr_0	C_BASEADDR	0x88208000	0x88209fff	8K	SPLB
push	C_BASEADDR	0x81400000	0x8140ffff	64K	SPLB
dip	C_BASEADDR	0x81420000	0x8142ffff	64K	SPLB
LEDs_8Bit	C_BASEADDR	0x81440000	0x8144ffff	64K	SPLB
xps_intc_0	C_BASEADDR	0x81800000	0x8180ffff	64K	SPLB
delay	C_BASEADDR	0x83c00000	0x83c0ffff	64K	SPLB
RS232_DCE	C_BASEADDR	0x84000000	0x8400ffff	64K	SPLB
DDR_SDRAM	C_MPMC_BASEADDR	0x8c000000	0x8fffffff	64M	SPLB0

Figure 1-60. Generate Addresses for Interrupt Controller and Timer peripherals

- ❺ In the **Ports** section, type in **timer1** as the **Interrupt** port connection of the **delay** instance, and hit enter.
- ❻ Make a new net connection (see **Figure 1-61**) for the **INTERRUPT** (external interrupt request) port on the **microblaze_0** instance by selecting **New Connection** from the drop-down box. This will create a net called **microblaze_0_INTERRUPT**.

Bus Interfaces		
Ports		
Addresses		
Name	Net	Direction
External Ports		
microblaze_0		
MB_Halted	No Connection	0
DBG_STOP	No Connection	1
INTERRUPT	microblaze_0_INTERRUPT	1
MB_RESET	mb_reset	1
dlmb		
ilmb		

⑧ **Figure 1-61. Make a new net connection to connect the MicroBlaze Interrupt port**

- ⑦ Connect the interrupt controller and timer as follows (refer to **Figure 1-60**)
 - Connect interrupt output port **Irq** of the **xps_intc_0** instance to the MicroBlaze interrupt input port using the *microblaze_0_INTERRUPT* net.
 - Click in **intr** field of **xps_intc_0** field to open the **Interrupt Connection Dialog**. Click on **timer1** on left side, and click on  sign to add to the **Connected Interrupts** field (right), and then click **OK**.

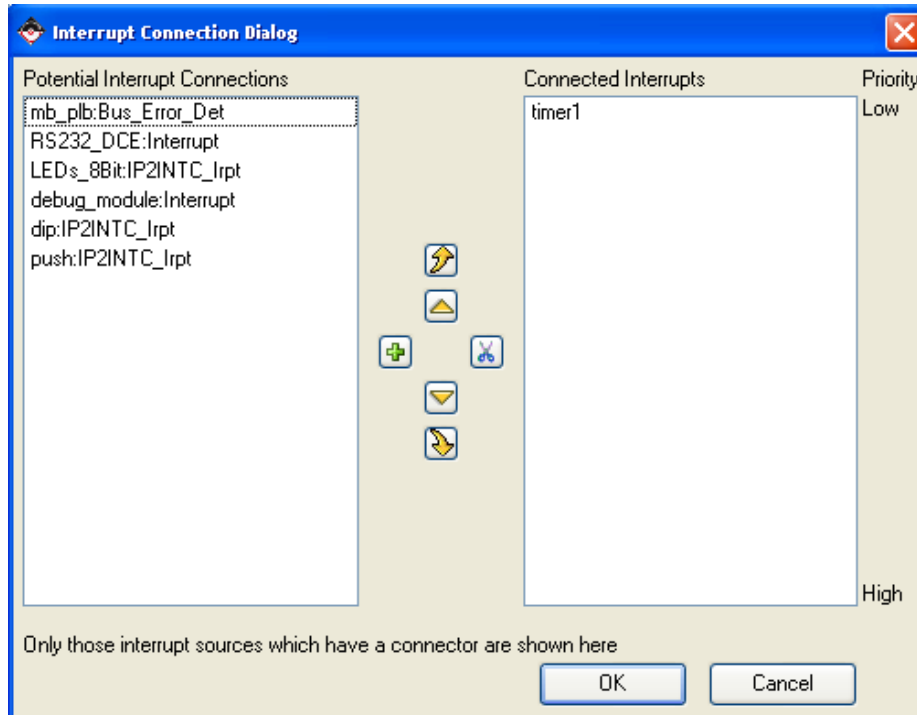


Figure 1-62. Connecting the Timer and Interrupt Controller

- Change the net name of **CaptureTrig0** port of **delay** instance to *net_gnd*.

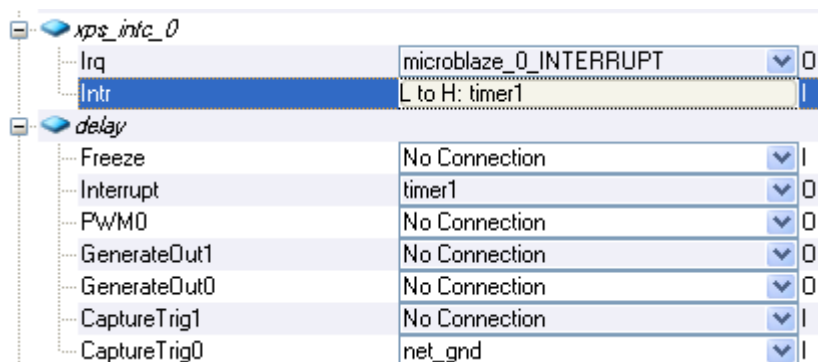


Figure 1-62. Connections Snapshot between Timer and Interrupt Controller

- ⑧ Double-click on **delay** to open its parameters box and check **Only One Timer is Present**. Click **OK** to accept the changes and close the dialog box
- ⑨ Select **Hardware → Generate Bitstream**

Create an SDK Software Project

Step 14



Launch SDK and create a new software application project for the lab XPS project. Import the lab_sdk.c source file.

- ❶ Open SDK by selecting **Software** → **Launch Platform Studio SDK**
- ❷ Select **Import XPS Application Projects** and click **Next**.

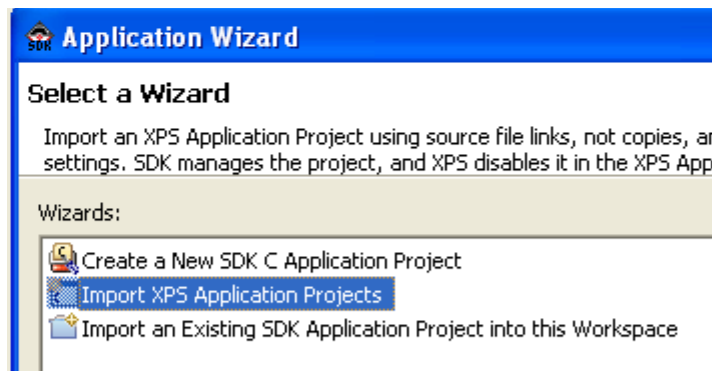


Figure 1-62. Managed Make C Project

- ❸ Put a check mark next to **TestApp_Memory** and click **Finish**.

This creates the directory **SDK_Projects/TestApp_Memory**, which is a copy of the TestApp_Memory software application project that was originally created with Base System Builder.

- ❹ Add **lab_sdk.c** by selecting **File** → **Import**. In the Import wizard, Double-click on **File System** and browse the directory to select them. Check the **lad_sdk.c** source file and click **Finish** to add the file to the project. For **Into Folder**, browse to and select **TestApp_Memory**. Click **Finish**

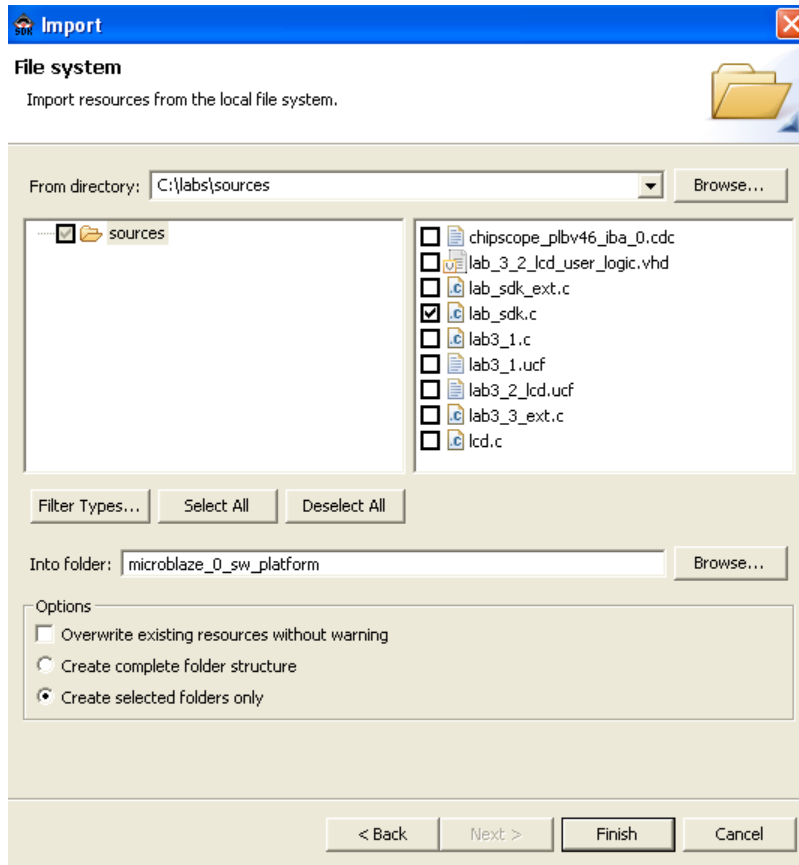


Figure 1-63. Importing Source Code

- ⑥ In the left hand **Navigators** tab, double click on the **lab_sdk.c** file to open it in the editor. The file is built as soon as it is opened, and note that both the **Problems** and **Console** tabs on the bottom report several compilation errors. The project is automatically built each time files in the project are edited and saved. Note also that the project outline on the right side is updated to reflect the libraries and routines used in the source file
- ⑦ In the **Problems** tab, double-click on the second red **x** for the parse error. This will bring you around to the line 86.

```

79  /* Initialize and set the direction of the GPIO connected to
80  XGpio_Initialize(&gpio, XPAR_LEDS_8BIT_DEVICE_ID);
81  XGpio_SetDataDirection(&gpio, LEDChan, 0);
82
83  /* Start the interrupt controller */
84  XIntc_mMasterEnable(XPAR_XPS_INTC_O_BASEADDR);
85  XIntc_mEnableIntr(XPAR_XPS_INTC_O_BASEADDR, 0x1);
86
87  /* Set the gpio as output on high 8 bits (LEDs) */
88  XGpio_mSetDataReg(XPAR_LEDS_8BIT_DEVICE_ID, LEDChan, ~count);
89  xil_printf("The value of count = %d\n\r", count);

```

Figure 1-64. First Error

- ⑧ Add the missing global variable declaration as **unsigned int**, initialize it to the value of 1, and save the file. The first error message should disappear.
- ⑨ Click the next error message to highlight the problem in the source code

```

100
101 /* Start the timers */
102 XTmrCtr_mSetControlStatusReg(XPAR_DELAY_BASEADDR, 0, XTC_CSR
103                               XTC_CSR_AUTO_RELOAD_MASK | XTC_CSR_DOW
104 /* Enable MB interrupts */
105 //microblaze_enable_interrupts();
106
107 /* Wait for interrupts to occur */
108 while(1) {
109     if(one_second_flag){
110         count_mod_3 = count % 3;
111         if(count_mod_3 == 0)
112             xil_printf("Interrupt taken at %d seconds \n\r",co
113         one_second_flag=0;
114         xil_printf(".");
115     }
116 }
117 }

```

Figure 1-65. Second Error

- ⑩ Add the missing global variable declaration as **int**, initialize it to the value of 0, and save the file. The additional error messages should disappear.

Write an Interrupt Handler

Step 15



Create the interrupt handler for the XPS timer. This source code is defined from `lab_sdk_ext.c`

- ① Go to where the interrupt handler function has already been stubbed out in the source file (a fast way to do this is to double-click on the function in the outline view).
- ② Create new local variable for the **timer_int_handler** function:

```
unsigned int csr;
```



The first step in creating an XPS timer interrupt handler is to verify that the XPS timer caused the interrupt. This can be determined by looking at the XPS Timer Control Status Register. Open the API documentation to determine how the Control Status Register works.

- ① In the **XPS System Assembly View** window, right-click the **delay** instance and select **View PDF Datasheet** to open the data sheet
- ② Go to the **Register Description** section in the data sheet and study the **TCSR0** Register. Notice that bit 23 has the following description:

Timer0 Interrupt

Indicates that the condition for an interrupt on this timer has occurred. If the timer mode is capture and the timer is enabled, this bit indicates a capture has occurred. If the mode is generate, this bit indicates the counter has rolled over. Must be cleared by writing a 1

Read:

0 - No interrupt has occurred

1 - Interrupt has occurred

Write:

0 No change in state of T0INT

1 Clear T0INT (clear to '0')

- ③ The level 0 driver for the XPS timer provides two functions that read and write to the Control Status Register. View the timer API doc by right-clicking on the **delay** instance in the System Assembly View and selecting **Driver:tmrctr_v1_10_b → View API Documentation**. In the API document, click on the **File List** link at the top of the document, then click on the link labeled **xtmrctr_1.h** in the file list. This brings up the document on identifiers and the low-level driver functions declared in this header file. Scroll down in the document and click on the link for the **XTmrCtr_mGetControlStatusReg()** function to read more about this function. Use this function to determine whether an interrupt has occurred. The following is the pertinent information found in the XPS timer documentation:

XTmrCtr_mGetControlStatusReg (BaseAddress, TmrCtrNumber)

Get the Control Status Register of a timer counter

- **Parameters:**

- **BaseAddress** is the base address of the device.
- **TmrCtrNumber** is the specific timer counter within the device, a zero-based number, 0 -> (XTC_DEVICE_TIMER_COUNT - 1)

- **Returns:**

- The value read from the register, a 32-bit value

- ③ Add the **XTmrCtr_mGetControlStatusReg** function call to the code with the associated parameters. The resulting 32-bit return value should be stored in the variable **csr**.

```
csr = XTmrCtr_mGetControlStatusReg(baseaddr, 0);
```

Note: Substitute **baseaddr** with the base address for the **delay** peripheral. Refer to **xparameters.h**

- ④ Complete the Interrupt handler (see **Figure 1-66**) according to the steps below

1. Test to see if bit 23 is set by ANDing **csr** with the **XTC_CSR_INT_OCCURED_MASK** parameter.
2. Increment a counter if an interrupt was taken.
3. Display the count value by using the **LEDs_8Bit** peripheral and print the value using **xil_printf** (same functionality as **printf** with the exception of floating-point handling)

Hint: You may use the **XGpio_DiscreteWrite ()** function

4. Clear the interrupt by using the following function call:

```
XTmrCtr_mSetControlStatusReg(baseaddr, 0, csr);
```

```

53 void timer_int_handler(void * baseaddr_p) {
54     /* Add variable declarations here */
55     unsigned int csr;
56
57     /* Read timer 0 CSR to see if it raised the interrupt */
58     csr = XTmrCtr_mGetControlStatusReg(XPAR_DELAY_BASEADDR, 0);
59
60     /* If the interrupt occurred, then increment a counter and set one_second_flag */
61     if (csr & XTC_CSR_INT_OCCURED_MASK)
62     {
63         count++;
64         one_second_flag = 1;
65     }
66
67     /* Display the count on the LEDS and print it using the UART */
68     XGpio_DiscreteWrite(&gpio, LEDChan, count);
69     xil_printf("Count value is: %x\r\n", count);
70
71     /* Clear the timer interrupt */
72     XTmrCtr_mSetControlStatusReg(XPAR_DELAY_BASEADDR, 0, csr);
73
74 }

```

Figure 1-66. Completed Interrupt Handler Code

- ⑤ Save the file, this should compile the source successfully.

Add Linker Script

Step 16



Remove TestApp_Memory_linker_script.ld file. Assign lab_sdk_LinkScr.ld as the linker script for building the project and compile the application.

- ① Click on the C/C++ Projects tab on the left side
- ② Right-click on **TestApp_Memory** and select **Properties** to open the **Properties** dialog box (or select **Project** → **Properties** from the menu)

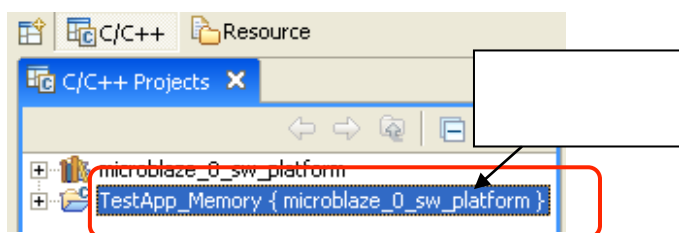




Figure 5-11. Software Project

- ③ In the left hand window of the **Properties** dialog, select the **C/C++ Build** item
- ④ Select the **Linker Script** option and click the **delete** button () to remove the TestApp_Memory_linker_script.ld script file.
- ⑤ Click the **Add** button () and add the **lab_sdk_LinkScr.ld** file (Figure 1-68).

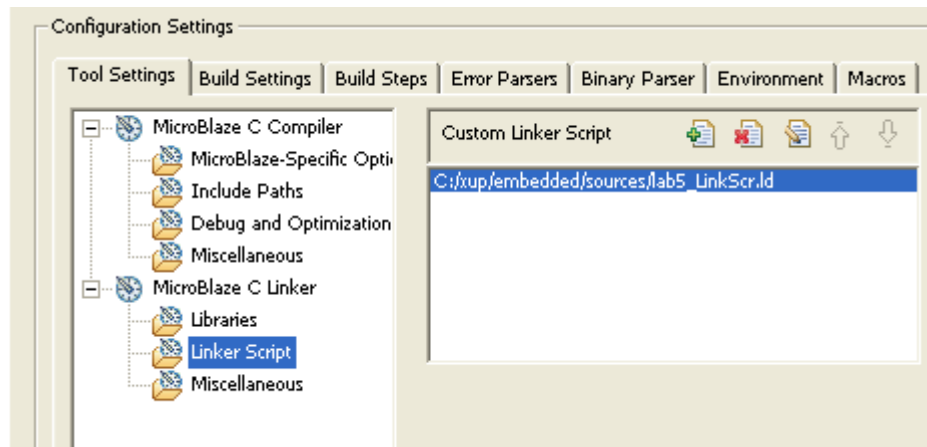


Figure 1-68. Adding Linker Script

- ⑥ Click **OK** to exit the **Properties** dialogue which will also recompile the program.

Verify Operation in Hardware

Step 17



Generate the bitstream and download to the Spartan-3E starter kit.

- ① Connect and power the board
- ② Select **Device Configuration** → **Program FPGA**
- ③ Select **TestApp_Memory.elf** the Initialization ELF.

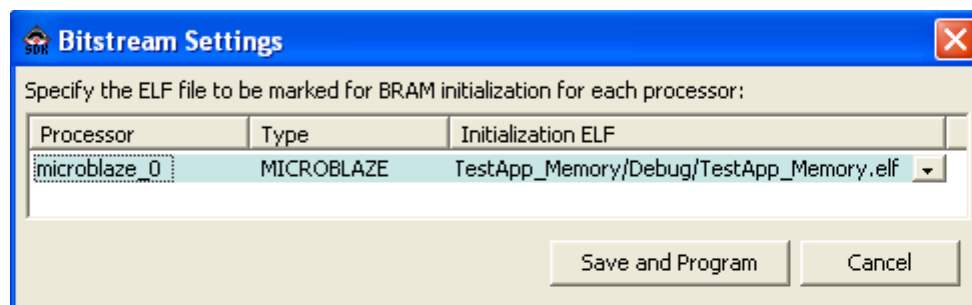


Figure 5-13. Selecting executable for BRAM initialization

- ④ Click **Save and Program**

This will configure the FPGA and you should observe a message on the hyperterminal window indicating the count value. The LEDs should be flickering.


```

The value of count = 1
.Count value is: 2
.Count value is: 3
Interrupt taken at 3 seconds
.Count value is: 4
.Count value is: 5
.Count value is: 6
Interrupt taken at 6 seconds
.Count value is: 7
.Count value is: 8
.Count value is: 9
Interrupt taken at 9 seconds

```

Figure 1-70. HyperTerminal Output

Debugging Using SDK

Step 18



Configure Target Connection Settings

- ❶ On the SDK Menu, select **Run → Run...**

This will present a screen summarizing the existing Launch Configurations

- ❷ Under Configurations, select **Xilinx C/C++ ELF**

- ❸ Click on **New** to add a new Launch configuration.

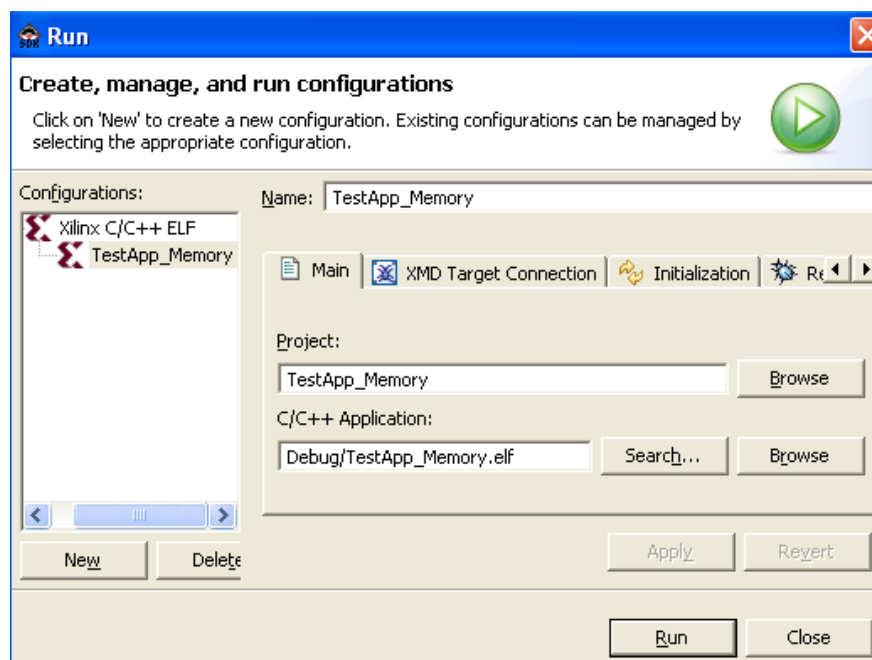


Figure 1-71. Setting Up Run Configuration

- ❹ Click on the **Run** button to establish a connection between the debugger and hardware target

You should see output displayed on hyperterminal since the program is running.

- ⑤ In the XMD Console view, type 'stop' at the XMD% prompt to stop the running process (Figure 1-72)

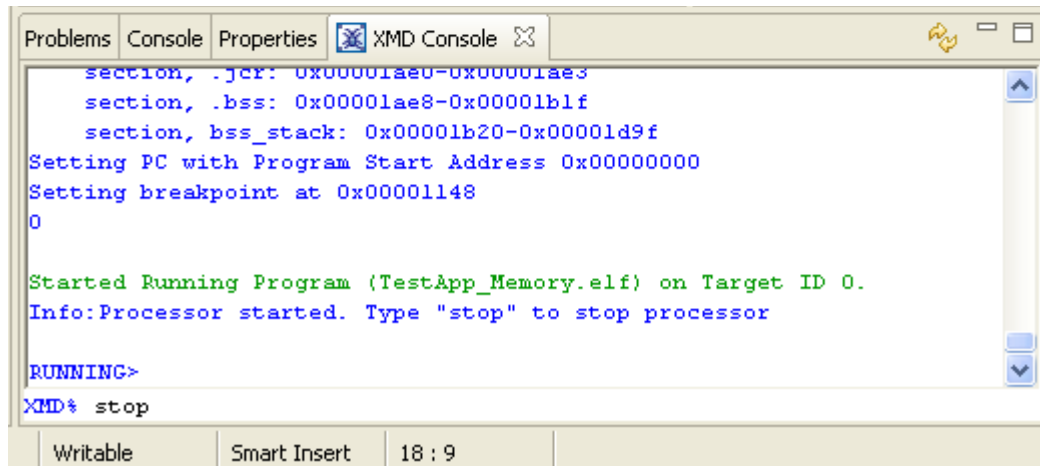


Figure 1-72. SDK's XMD Console



Launch Debugger and debug.

- ① On the SDK Menu, select **Run → Debug...**

This will present a screen summarizing the existing Launch Configurations

- ② Click on **Debug**. If a dialog box appears asking you to confirm whether to switch to the Debug Perspective, click **Yes**

This opens the Debug perspective. The debugger is automatically connected to the processor via XMD. The processor will be suspended automatically (breakpoint) at the first statement in main()

- ③ Click on the **Resume** button. The application will run

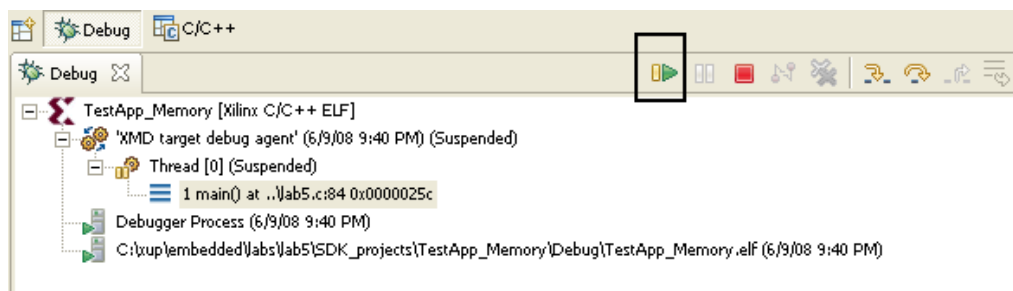


Figure 1-73. Resuming an Application

- ④ Click on the **Thread[0] (Running)** line in the **Debug** window (left) and click the **Suspend** button to suspend operation.

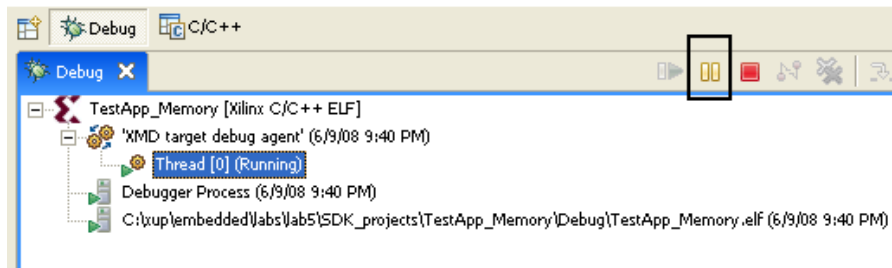


Figure 1-74. Suspending a Running Application

- 5 Right click in the **Variables** tab and select **Add Global Variables ...** All global variables will be displayed. Select **count** variable and click **OK**
- 6 Right click on **count** and make sure that **Enable** is selected



Monitor variables and memory content.

- 1 Double-click to set a breakpoint on the line in **lab_sdk.c** where count is written to LED

```

56
57 /* Read timer 0 CSR to see if it raised the interrupt */
58 csr = XTmrCtr_mGetControlStatusReg(XPAR_DELAY_BASEADDR, 0);
59
60 /* If the interrupt occurred, then increment a counter and set one_second_flag */
61 if (csr & XTC_CSR_INT_OCCURED_MASK)
62 {
63     count++;
64     one_second_flag = 1;
65 }
66
67 /* Display the count on the LEDs and print it using the UART */
68 XGpio_DiscreteWrite(&gpio, LEDChan, count);
69 xil_printf("Count value is: %x\r\n", count);
70

```

Figure 1-75. Setting Breakpoint

- 2 Click on **Resume** button to continue executing the program up until the breakpoint.

As you do step over, you will notice that the **count** variable value is changing.

- 3 Click on the memory tab. If you do not see it, go to **Window → Show View → Memory**
- 4 Click the + sign to add a Memory Monitor

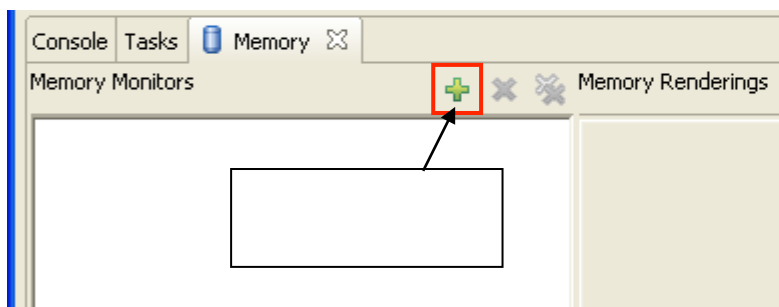


Figure 1-76. Add Memory Monitor

- 5 Enter the address for the **count** variable as follows, and click OK

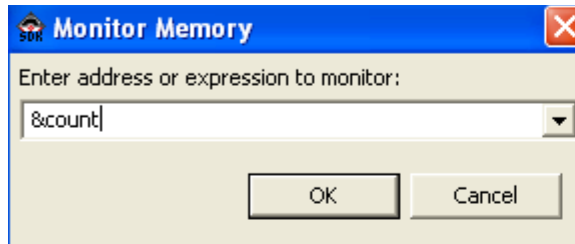


Figure 5-21. Monitoring a Variable

- ⑥ Click the **Resume** button to continue execution of the program.

Notice that the count variables increment every time you click resume.

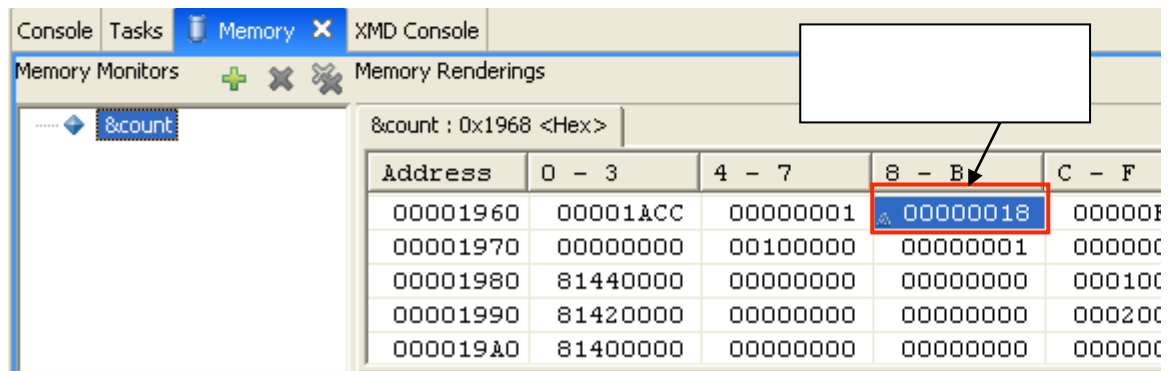


Figure 1-77. Viewing Memory Content of the count variable

- ⑦ Terminate the session by clicking on the **Terminate** button.

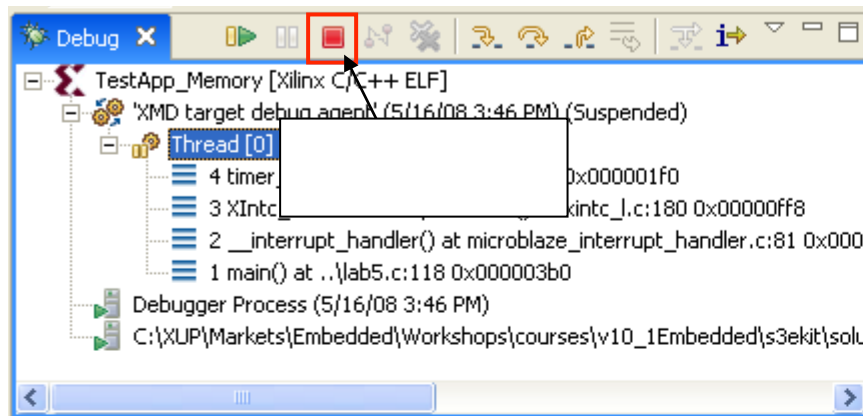


Figure 1-78. Terminating a Debug Session

Close the SDK application

HW/SW System Debug Lab: MicroBlaze

Step 18

You will extend the system created in the previous lab by adding Chipscope ICON and IBA cores. The IBA core will be added to the PLB bus. You will set trigger conditions in the Chipscope Analyzer software (running on PC) to capture bus transactions when the value of the count variable is written to the LEDs. When the hardware trigger condition is met, you will see that the software debugger stops at the line of code that was last executed.

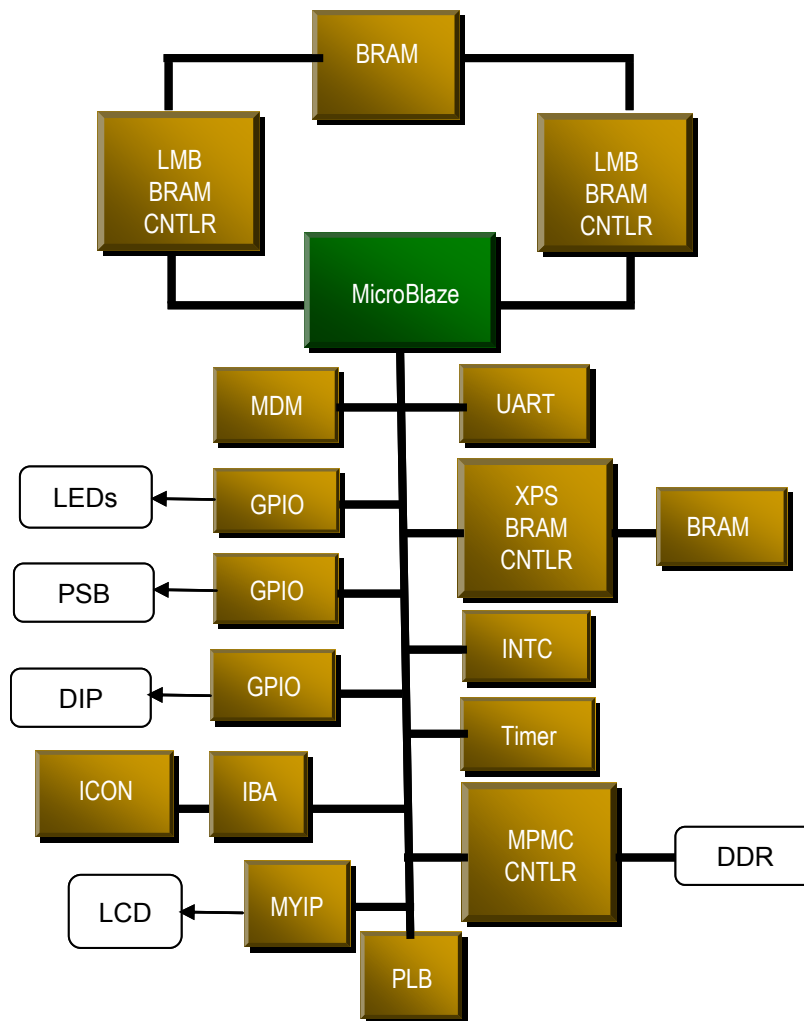


Figure 1-79. Complete MicroBlaze System

Instantiate ChipScope Cores

Step 19



Add the ChipScope cores using the Debug Configuration wizard. Configure the device and the design to the following ports, as shown in the Figure 1-80. Setup the trigger to trigger when a certain values are on the PLB address, PLB data, and PLB control bus.

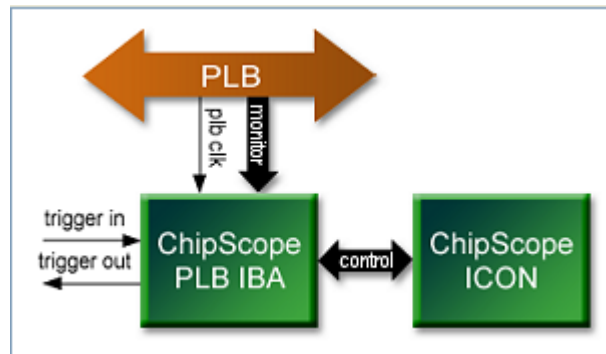


Figure 1-80. ChipScope Core Connections

❶ Select **Debug** → **Debug Configuration**

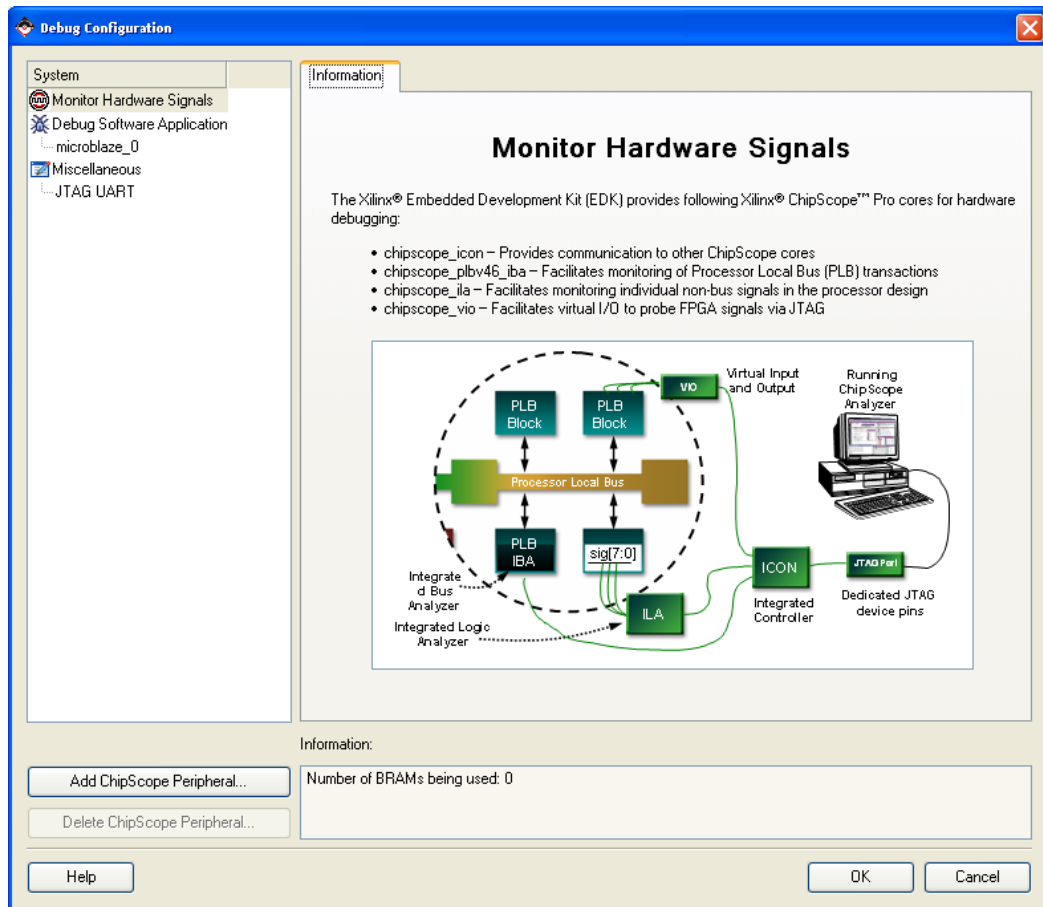


Figure 1-81. Debug Configuration Dialogue

❷ Click the **Add ChipScope Peripheral...** button and select the first option, **To monitor PLB v4.6 bus signals (adding PLB IBA)**. Click **OK**.

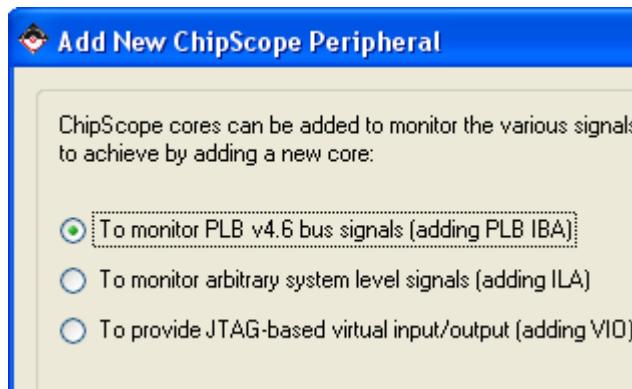


Figure 1-82. Add the PLB IBA

- ③ Click to put a check mark in the **Bus Write Data Signals** field and set the **Select the Number of signal samples you want to collect** option to **512**. Make sure you have the options selected according to **Figure 1-83**.

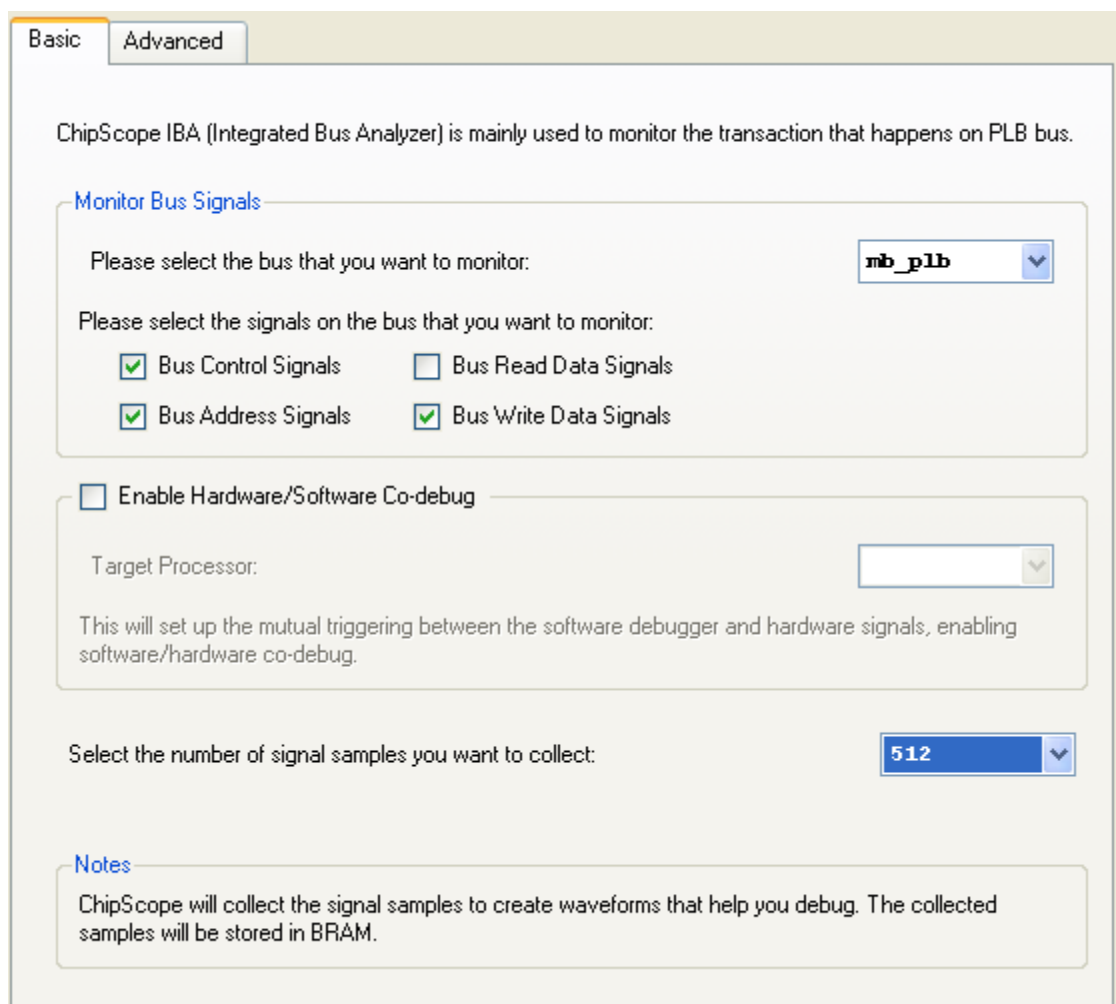


Figure 1-83. Setting Basic Debug Configuration Options for the PLB_IBA

- ④ Click the **Advanced** tab. Under the **User** tab, in the **Trigger In, PLB Reset and PLB Error Status** panel, uncheck the **Enable probing system reset and system error signals** field and set **Match unit type** to **basic**

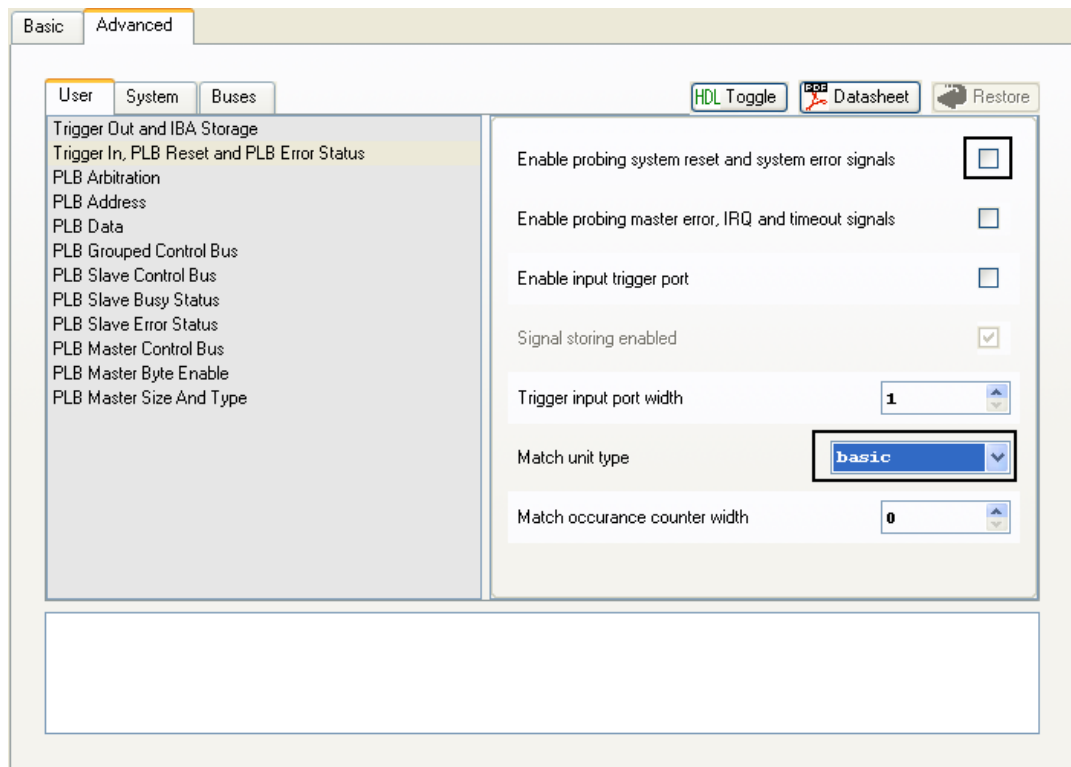
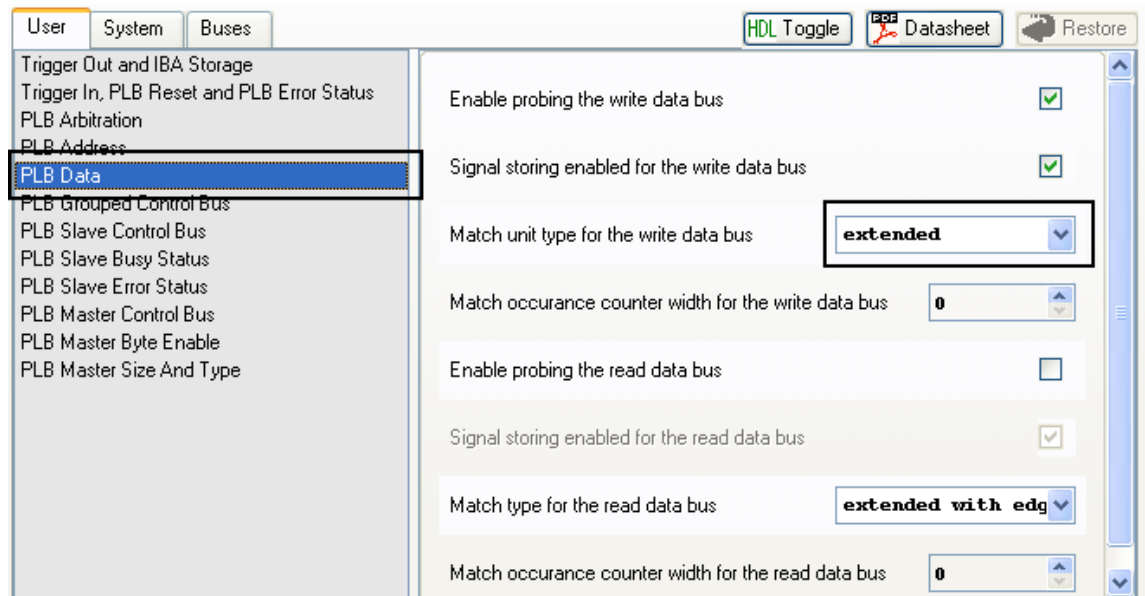


Figure 1-84. Setting Trigger In, PLB Reset and PLB Error Status options

- ⑤ Select **Extended** as the **Match Unit Type** for the **PLB Address** and **PLB Write Data** busses



- ⑥ Click **OK**, and view the Bus Interface noting the newly added Chipscope Cores in the System Assembly View

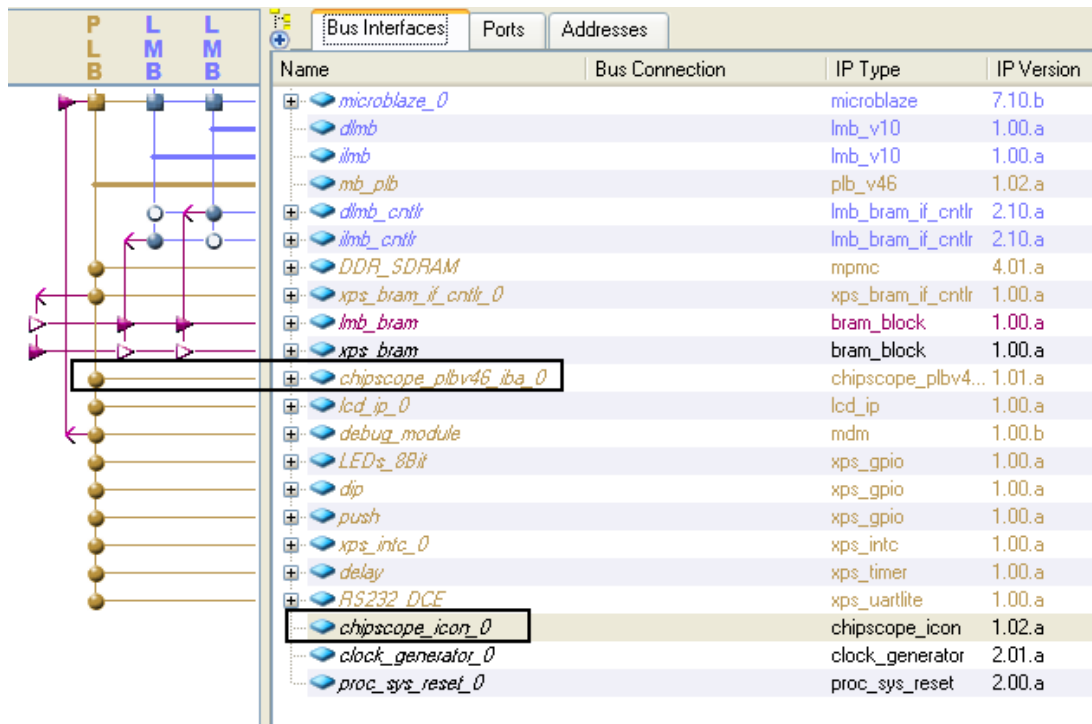


Figure 1-85. Chipscope Cores Automatically added to MicroBlaze System

- 8 Select **Hardware** → **Generate Bitstream**

Setup SDK and ChipScope

Step 20



Open an SDK project and establish a connection to the target using XMD. Having successfully generated your design it is possible to begin viewing it in operation using the **SDK debugger** and **ChipScope Pro** tools.

Starting the SDK debugger (Software Debug)

- 1 Launch SDK: select **Software** → **Launch Platform Studio SDK**
- 2 Click **cancel** when the wizard opens
- 3 Delete the existing project that was created in Lab (right-click on TestApp_Memory and select delete – **do not delete contents**)
- 4 Import the lab_sdk project: go to **File** → **Import** → **Existing Projects into Workspace** and browse to the lab\SDK_projects\TestApp_Memory. Click **OK**.

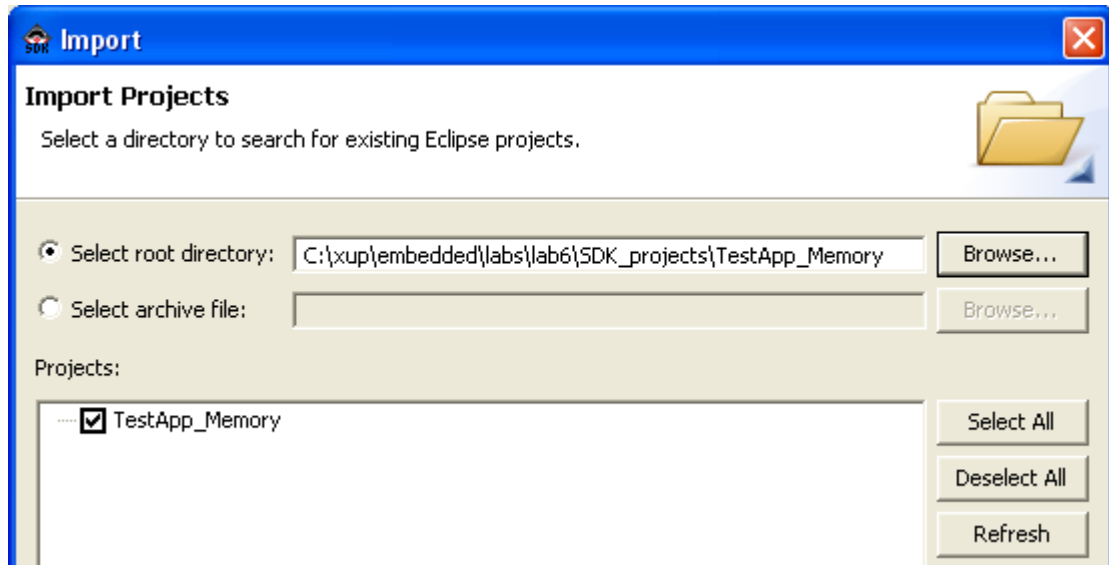


Figure 1-86. Importing an Existing Project into SDK

- 5 With **TestApp_Memory** selected under **Projects**, click **Finish**.
- 6 With the board connected and powered, select **Device Configuration** → **Program FPGA** to update the bitstream with the executable and download the bitstream to the FPGA.

Operation should still be the same.

- 7 Setup the target XMD connection by selecting **Run** → **Run...**, and click the **Run** button

Operation should still be the same.

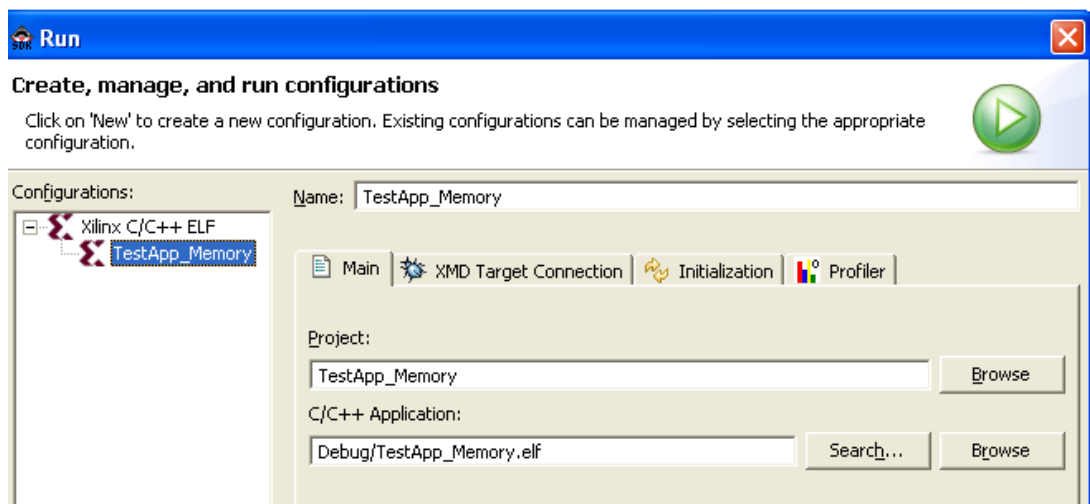


Figure 1-87. Specify Project and .elf Location

- 8 Invoke the debugger by selecting **Run** → **Debug...**, and then click the **Debug** button.

The SDK Debugger should now be connected to the target and operation should be suspended (**Figure 1-88**). Code operation will be halted at the first line following the `main()` routine.

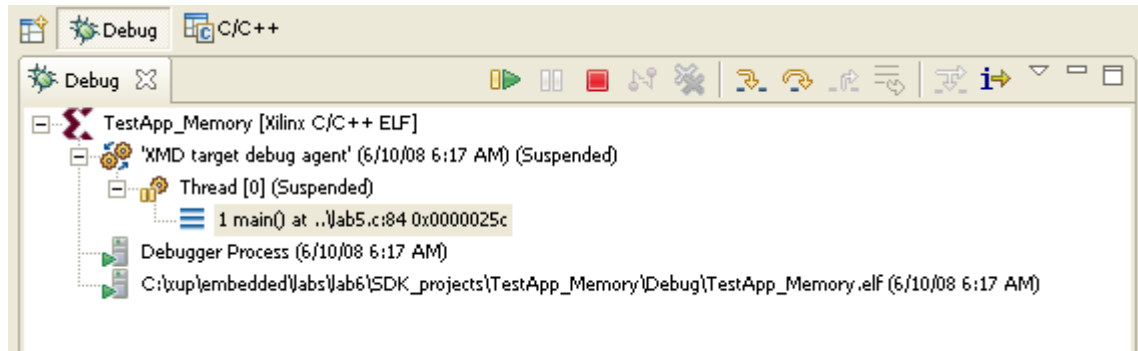



Figure 1-88. SDK Debugger Connected to Target via XMD



Starting ChipScope Pro (Hardware Debug)

- ❶ Launch the ChipScope Pro Analyzer tool from the program group or desktop icon
- ❷ Click on the Open Cable/Search JTAG chain icon.  This will identify the devices on the JTAG chain (**Figure 1-89**). Click **OK** to open ChipScope Pro Analyzer with default **Trigger Setup** and **Waveform signal** windows.

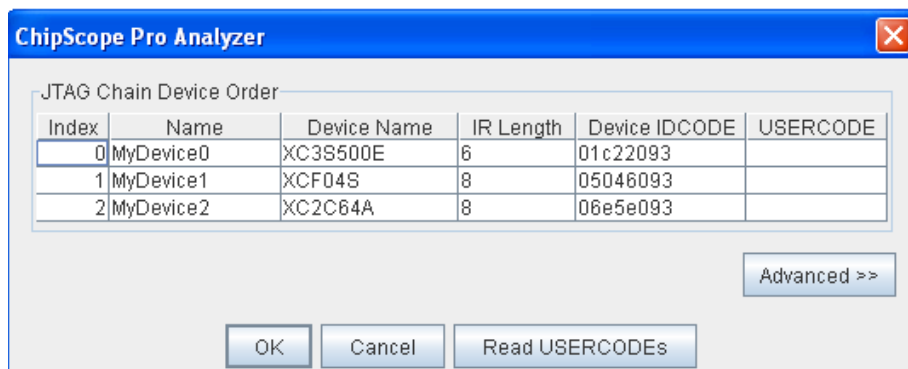


Figure 1-89. ChipScope JTAG Device Order

- ❸ Select **File → Import**. In the **Signal Import** dialogue click on the **Select New File** button.
- ❹ Browse to the sources directory and the select the following chipscope definition and connection file (CDC) `C:\lab\chipscope_plbv46_iba_0.cdc` and click **OK**.
The CDC file contains signals associated with the PLB core which should now be listed in the **Trigger Setup** and **Waveform signal** windows.

Perform HW/SW Verification

Step 21



Setup the trigger to capture 32 data samples when count values greater than 5 are written to the LEDs.

- ❶ Set **M0:TRG0 PLB_RNW** bit == 0 by clicking the + sign under **M0** and selecting the **PLB_RNW** bit and changing its value to 0 under **Value** field

- ② Change the Radix of **M1** and **M2** from binary (Bin) to Hexadecimal (Hex) by clicking on the respective boxes and selecting Hex
 - ③ Set **M1:TRIG1** == **8144_0000** (or base address of LEDs_8Bit peripheral) and **M2:TRIG2** > **0000_0005** by selecting and adjusting the value box
 - ④ Click the field under **Trigger Condition Equation**, which opens the **Trigger Condition: TriggerCondition0** dialog box. Select **M0** and Select **M1**, and then click **OK** to close.
- The **Trigger Condition Equation** field should now display **M0 && M1**. Click **OK**.
- ⑤ Set the trigger window depth to **32** and position to **0**
 - ⑥ Set the **Storage Qualification** (**M0 && M1 && M2**) so that you capture count values greater than 5 when written to the LEDs_8Bit peripheral.

Your settings should be similar to **Figure 1-90**.

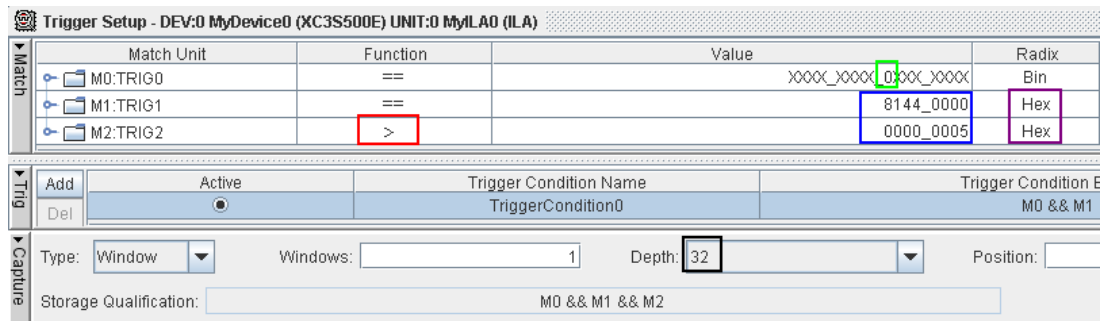


Figure 1-90. Chipscope Trigger Settings

- ⑦ Delete all the signals from the **Waveform** window using [Shift Select] except for **PLB_RNW**.
- ⑧ In the **Signals** window, select **PLB_ABus** and **PLB_wrDBus** and add them to the waveform by right click on each bus and selecting **Add To View** (see **Figure 1-91**).

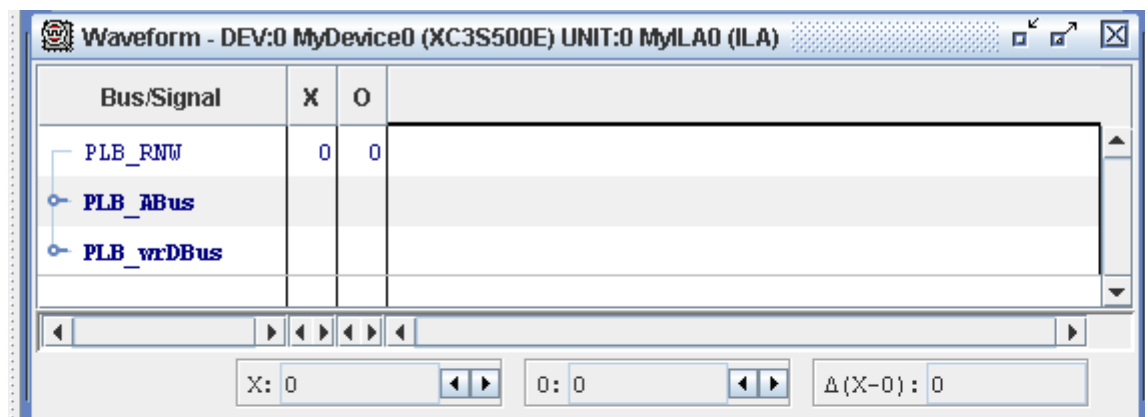


Figure 1-91. Chipscope Waveform View Setup

- ⑨ Setup the trigger by selecting **Trigger Setup** → **Run**.



Run Software debugger and wait for the condition to trigger

- ❶ In software debugger window (opened before) click on **Resume** to continue with debug.

The ILA core will trigger when a value greater than 5 is written to the LEDs. The buffer will be filled with 32 data samples, which will be displayed in Chipscope-Pro Analyzer (see **Figure 1-92**).

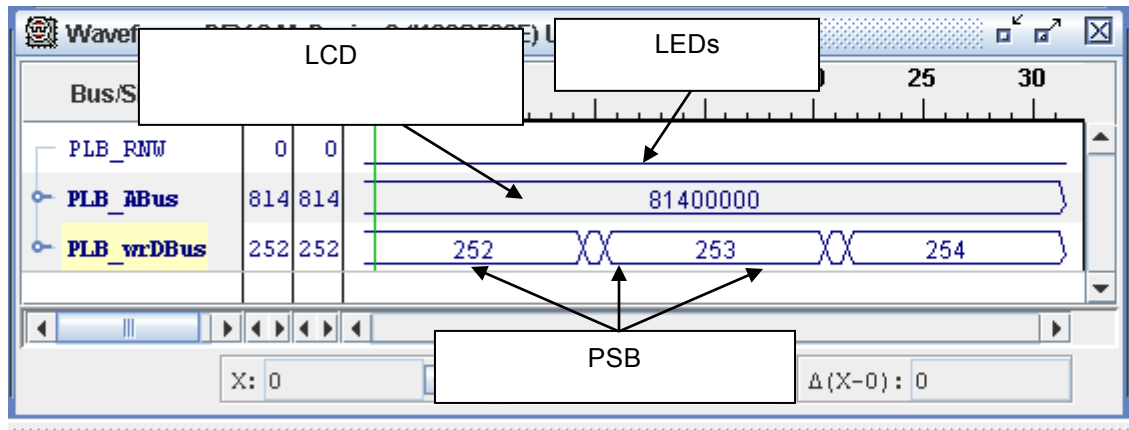


Figure 1-92. Chipscope-Pro Debug Results

- ❷ Stop the debugger in SDK
- ❸ Close SDK, XPS, and ChipScope programs